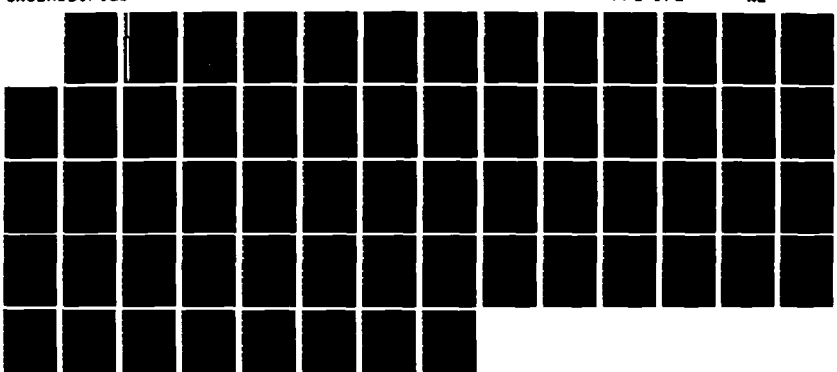
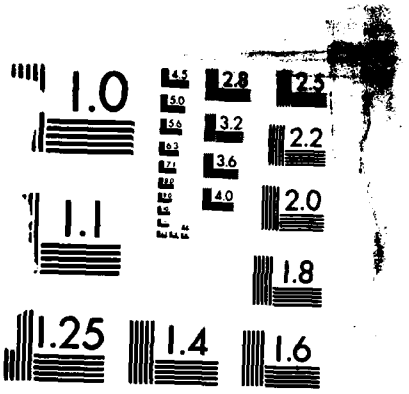
AD-A165 893 A DISCIPLINE FOR LOOP CONSTRUCTION(U) WOLLONGONG UNIV 1/1
(AUSTRALIA) DEPT OF COMPUTING SCIENCE R G DROMEY
21 JAN 83 83-1 R/D-4278-CC DAJA45-83-M-0046

UNCLASSIFIED F/G 9/2 NL





THE UNIVERSITY OF WOLLONGONG

DEPARTMENT OF COMPUTING SCIENCE

A DISCIPLINE FOR LOOP CONSTRUCTION

R. Geoff Dromey

Department of Computing Science
University of Wollongong

Abstract

A discipline for loop construction is presented which is based on the concept of a well-formed postcondition. A well-formed postcondition is seen to have an implicit logical structure which is made explicit by appropriate variable binding. This variable binding identifies the loop invariant and a determinate. Loops are then constructed by first identifying the weakest iterative mechanism capable of establishing the postcondition. Subsequent development proceeds by way of inductive stepwise refinement. This discipline for loop construction leads naturally to a scheme for classifying loop mechanisms. It also leads to a proposal for a weak loop grammar (not in principle unlike Chomsky's phrase structure grammar) which helps to make explicit semantically important components of a loop structure. The grammar is enhanced by a set of fundamental transformation rules.

Preprint No 83-1 January 21, 1983

DAJA45-83-M-0046

P.O. Box 1144, WOLLONGONG, N.S.W. AUSTRALIA
telephone (042)-282-981
telex AA29022

FINAL
~~DRAFT ONLY~~

21 Jan '83

A DISCIPLINE FOR LOOP CONSTRUCTION

R. Geoff Dromey

Department of Computing Science,
The University of Wollongong,
Post Office Box 1144,
Wollongong, N.S.W. 2500
Australia.

ABSTRACT

A discipline for loop construction is presented which is based on the concept of a well-formed postcondition. A well-formed postcondition is seen to have an implicit logical structure which is made explicit by appropriate variable binding. This variable binding identifies the loop invariant and a determinate. Loops are then constructed by first identifying the weakest iterative mechanism capable of establishing the postcondition. Subsequent development proceeds by way of inductive stepwise refinement. This discipline for loop construction leads naturally to a scheme for classifying loop mechanisms. It also leads to a proposal for a weak loop grammar (not in principle unlike Chomsky's phrase structure grammar) which helps to make explicit semantically important components of a loop structure. The grammar is enhanced by a set of fundamental transformation rules.

FINAL
~~DRAFT ONLY~~

21 Jan '83

A DISCIPLINE FOR LOOP CONSTRUCTION

R. Geoff Dromey

Department of Computing Science.
The University of Wollongong.
Post Office Box 1144.
Wollongong, N.S.W. 2500
Australia.

1. Introduction

"Everything should be made as simple as possible, but not simpler"

Einstein

Loops play a central role in programming and therefore deserve a sound methodology to support their development and application. The work of Floyd on assertions [1], Hoare on loop invariants [2], and Dijkstra's weakest precondition methodology [3] has gone a long way towards formalizing certain aspects of the construction and characterization of loops. Despite this formalization, and the positive contribution of structured programming to loop construction it is apparent that there are still some aspects of the methodology that require further clarification and development.

We suggest that existing methodology and practice does not always provide adequate guidance on the most appropriate initialization for a loop, on what constitutes a suitable guard for a loop, on how the loop invariant can be determined, and on where various constructs should be arranged in the body of a loop. Imprecision with any of these details can often result in poorly constructed loops. In the discussion which follows we will attempt to address and clarify a number of these issues.

Dijkstra's methodology [3] provides a useful reference point from which to view our alternative proposal. We will therefore begin our discussion by briefly reviewing his method.

The starting point with Dijkstra's method is a formal specification of the problem. This specification must yield an associated postcondition R which precisely defines the goal to be accomplished. If it is then recognized that the problem requires a loop (or loops) the next step is to determine an appropriate loop invariant P by weakening (or generalizing) the predicate R in some way. Useful heuristics which have been suggested for this purpose include, replacing a constant by a variable in R, deleting a conjunct from R, and adding a disjunct to R. An initialization process is then performed to establish the truth of the invariant P before the loop is entered. The guard B for the loop is then developed so that $P \wedge \neg B \Rightarrow R$. Finally, the body of the loop is developed in such a way that it will decrease a bound function associated with the loop with each iteration while re-establishing the truth of the invariant. Detailed accounts and applications of this methodology are given in monographs by Dijkstra [3] and Gries [4].

At this point we have not considered alternative programming disciplines as suggested by Jackson [5] and Warnier [6]. Later in the discussion some of the issues raised by Jackson's method in particular will be taken into account. With the background we have outlined we can now consider an alternative strategy for loop

construction. In presenting our proposals we have deliberately chosen to keep the treatment relatively informal to minimise distraction from clearly presenting how the method can be applied in practice in a straightforward manner.

In presenting his methodology Dijkstra suggests a number of ways of weakening the postcondition R to obtain the invariant P. However in practice the problem of determining an appropriate loop invariant prior to constructing a loop often requires considerably more invention, insight, and experience, than is perhaps desirable.

We would suggest that the problem of determining an appropriate loop invariant can be eased by taking a somewhat different approach.

2. A Loop Calculus

A starting point for the discussion of an alternative methodology is to take a somewhat different view of the postcondition R to that conventionally used. There is clearly more than one way in which we can express a postcondition R for a particular problem. Furthermore some expressions R are more useful than others in conveying information that may be helpful in constructing the loop to establish the goal sought. In this respect the suggestion made by Dijkstra is to formulate a postcondition that will permit generalization (or weakening) in moving back further from the end-result. We would suggest that the process of characterizing useful postconditions can be taken a step further by requiring that the postconditions R always assume a specific logical structure. This structure requires that the postcondition R for a loop be thought of as the conjunction of an invariant *P* and a *determinate* D. i.e.

$$R = P \wedge D$$

A postcondition R with this structure is said to be *well-formed*. We will see a little later how the logical elements P and D are projected from well-formed postconditions R. The *invariant* P is that predicate component of R which is true before and after each loop iteration. The truth of the *determinate* D for some specific value or values of variables associated with R *determines* the truth of the postcondition R.

The role of the loop body when we have a well-formed postcondition is to establish the truth of the *determinate* while maintaining the truth of the *invariant* P. With the *determinate* D true the postcondition R is established. Progress towards establishing the truth of the determinate can be measured by associating with a loop a bound function t that remains greater than or equal to zero and which is monotonically decreasing with successive iterations of the loop.

The job of the guard B of the loop is to terminate the loop either when the determinate is true or when a state has been reached where it is possible to establish the truth of the determinate by some mechanism associated with, but external to the loop.

2.1. Projection of the Invariant and Determinate from the Postcondition

To do this we must examine the underlying nature of the postcondition R. Postconditions are expressed in terms of *variables* and *constants*. The constants represent the input or given information about the problem. The role of the loop or program that must be derived is to perform operations on the variables and the given data until a state is attained which satisfies the postcondition.

Initially when the postcondition is derived the free variables associated with the problem have not been initialized or bound to specific values although they may have attached ranges which define the set of possible values that they may assume.

With our proposed method for determining the loop invariant a first step is to identify the free variables in the postcondition. The next step is to attach valid ranges to each of these variables (as noted before, some may already have ranges attached

in the postcondition). In some instances it is possible to establish both a lower and an upper bound for a variable while in other cases it may only be possible to establish either a lower or an upper bound or some defined value. In such cases the undefined bound can only be expressed symbolically and is therefore not of any direct use in the steps to follow. In general the bounds for variables can be derived from a knowledge of the given information associated with the problem. Once ranges have been established for all variables the next step is to use this information to *project* the invariant P out of the postcondition R. Projection of the invariant from the postcondition is achieved by binding all the variables in the postcondition to one of their *defined* bounds or limiting values. When this has been done those components of the postcondition that are established by the variable binding to be logically *true* for *all* valid states defined by the postcondition are said to constitute the *projected invariant* P for the associated postcondition R. These values of the variables referred to as their *projection values* can obviously be used for the initialization of the loop that still needs to be derived.

In essence we have *reversed* the order of the sequence of steps suggested by Dijkstra [3]. Rather than first determining the invariant P, and then assigning values to variables to establish it to be true initially before the loop is entered, we take the steps in reverse order. That is, valid bounds are assigned to variables and then these variable bindings are used to identify the invariant logical elements in the postcondition. This achieves the same end result as Dijkstra's method – an invariant, and an appropriate initialization to establish the initial truth of the invariant before loop entry. However, there is usually a lot less invention required to establish ranges for variables which in any case should be already available if the postcondition is well-formed.

Those logical components of the postcondition that are not established by the variable binding to belong to the invariant are identified as belonging to the *determinate*. They are true for specific values of variables and constants associated with the problem. The truth of the determinate, however, determines the truth of the postcondition for some specific configuration of the state variables. There are two states where the determinate is true, a non-iterative state corresponding to the smallest problem that the mechanism can solve and the more general state that applies at the termination of the iterative process. It is important to analyse both of these states. Choosing values of the given data that establish the truth of the determinate (assuming *variable binding* has been previously applied) without iteration allows us to determine what we shall call the *postcondition guard* B_R of the loop. This *data sizing* defines the dimensions of the smallest valid problem for which the postcondition can be established to be true. Accordingly, a corresponding guard must be applied to protect the mechanism from data for which the postcondition is not defined.

The more general state in which the determinate is true can only apply after an iterative process has terminated. Having identified the invariant and the determinate the next step is to determine the structure of the loop body.

2.2. Construction of the Loop Body

The role of the loop body is clearly to take the state of the computation from that of initialization to a state where the postcondition is satisfied. The suggestion for developing the loop body made by Gries [4] is to construct it so that it decreases the bound function while re-establishing the loop invariant. While this is sound advice it would be useful to provide more detailed and more specific advice on such issues as what structure the loop body should take, what components should make up that loop body, and how they can be constructed.

The business of problem-solving can invariably be made easier if it is possible to break a problem down into a number of sub-problems that can be considered one at a time. In the present context we are looking for a systematic procedure that will break

the problem of developing the loop body into a set of well-defined and more manageable sub-problems. At the same time, what should always be an important consideration in computing is the development of efficient solutions to problems.

In our search for such a systematic procedure we must pay more detailed attention to the nature and structure of the postcondition. With many problems that require iterative solution there is often a whole spectrum of given initial data configurations that must be taken through to the postcondition. Furthermore, some initial configurations should require considerably less effort than others to establish the postcondition (e.g. we may have a set of data to be sorted in which only a very small fraction of the elements are out of order). We may ask at this stage what is the use of investigating such specialized initial configurations for the given data? As it turns out consideration of such limiting special cases is often very useful in constructing mechanisms to solve *general* problems. As a starting point for constructing the loop body it is useful to consider first valid configurations of the given data that should require the least effort to establish the postcondition. The corresponding weakest iterative mechanism capable of establishing the postcondition under such conditions will be referred to as the *Π-mechanism* associated with the loop.

The nature of the Π-mechanism and the role that it can play in the solution of a problem of course can vary considerably from problem to problem. There are, however, a number of important classes of Π-mechanism that are easy to recognize and apply.

As the Π-mechanism alone is usually not capable of establishing the postcondition directly for more general initial states (e.g. it may be required to sort random data) it must frequently be accompanied by some other called the *Δ-mechanism*. The role of the Δ-mechanism, at each application, is to change the state of the computation into a configuration where the Π-mechanism can again be applied. In some cases this involved a straightforward initialization while in other instances deductive steps must be applied.

It is important to understand the relationship of the Π-mechanism with the postcondition. *In essence the Π-mechanism corresponds to that mechanism which changes the least number of variables associated with the postcondition but still allows it to decrease the bound function and hence make progress towards establishing the postcondition.* Whenever the Π-mechanism reaches a state where other variables must be changed to make further progress, the Π-mechanism terminates and the Δ-mechanism must be applied.

We will return to the application of these ideas after we have discussed how loop guards can be derived.

2.3. Derivation of the Loop Guard

To derive the guard for a loop characteristics of the *loop body* must be related to the bound function in order to derive the most appropriate guard.

The characteristics of the loop body which we must take into account in deriving loop guards can be embodied in what we shall call the *iterative capacity* ψ of the loop. The maximum *iterative capacity of a loop, ψ_{max} is the maximum amount by which the elements of the loop body can decrease the bound function associated with the loop in a single iteration for the smallest general problem that requires iterative solution.* Many loops have a maximum iterative capacity ψ_{max} of 1, in which case the whole process of deriving the guard is straightforward. In instances where the iterative capacity is greater than one considerable care should be taken in choosing the guard for the loop.

A rule that can be widely applied in deriving the guard for the loop is to simply set up a relationship such that the bound function t for the loop is greater than or equal to

the maximum iterative capacity Ψ_{max} of the loop i.e.

$t \geq \Psi_{max}$.

The bound function can usually be derived directly from the invariant or determinate if the postcondition is well-formed. In special cases where a loop contains only other loops a different strategy for determining the loop guard must be used. These issues will be discussed later.

There is an important difference between the present approach to determining the loop guard and that of Dijkstra [3] where the guard is derived *before* rather than *after* the body of the loop has been determined. Incorporating information about the body of the loop in deriving the guard avoids the problem of *loop* overdesign. Loop overdesign occurs when a loop is allowed to do more iterations than are sufficient to solve the problem. When a guard allows this condition it is often necessary to employ additional guards within the loop to ensure maintenance of the invariant. We will come back to the problem of loop overdesign later. When specific problems which exhibit this characteristic are considered, we can see how the methodology we have outlined can be applied in practice.

3. Some Small Examples

In this section we will illustrate how the methodology we have outlined can be applied to some well-known problems. In the first several examples the weakest iterative mechanism solves the problem for all cases and consequently does not require generalization.

Example 3.1 Integer Square Root Approximation

Given an integer n stablish the largest integer "a" that is less than or equal to ?n. The postcondition for this problem can be written as:

$R: 0 \leq a^2 \leq n < (a+1)^2$

It is convenient to rewrite it as a set of conjunctions, i.e.

$R: 0 \leq a^2 \wedge a^2 \leq n \wedge n < (a+1)^2$

The only variable in the postcondition is "a". The range for this variable r_a is:

$r_a: 0 \leq a \leq n$.

Possible *projection values* are a = 0, and a = n. Choosing 0 as the projection value for "a" and substituting it into the postcondition yields:

$0 \leq a^2 \Rightarrow$ true

$a^2 \leq n \Rightarrow$ true

$n<(a+1)^2 \Rightarrow$ true for $n0 = 0$, false for $n > 0$

Remembering that the invariant P is that part of R established to be true by variable binding using the projection values we get as the projected invariant

$P: 0 \leq a^2 \wedge a^2 \leq n$

The remaining part of R constitutes the determinate D for the problem. We have

$D: n < (a+1)^2$

A suitable bound function for the problem can be derived from the invariant. In this case we have n $\geq$ a\u2\d and hence we can use

t: $n-a^2 \geq 0$

The task at hand now is to construct a loop that will establish the truth of the determinate while maintaining the truth of the invariant. Once the determinate is true the postcondition will be established. We can attempt to make progress towards establishing D by increasing "a" under the invariance of P. The loop body can therefore consist of the assignment a := a+1 which must be applied in this case until the determinate is true (i.e. until n < (a+1)\u2\d). However, for consistency, rather than applying the complement of the determinate directly, we will use the ideas described in the last section for deriving guards. We have

t: $n - a^2 \geq 0$ (from invariant)

ψ_{max}: $2a + 1$ (i.e. $(a+1)^2 - a^2$)

B: $t \geq \psi_{max}$

B: $n - a^2 \geq 2a+1$ on substitution

B: $n \geq (a+1)^2 \equiv (a+1)^2 \leq n$

Notice that we have arrived at the same guard as by simply complementing the determinate in this simple case. Our implementation therefore has the form:

do $(a+1)^2 \leq n \longrightarrow a := a+1$ od

It is interesting as an exercise to use the other possible projection value for "a" (i.e. a = n). Substituting this value for "a" into the postcondition R yields:

$0 \leq a^2 \Rightarrow$ true

$n < (a+1)^2 \Rightarrow$ true

$a^2 \leq n \Rightarrow$ true for $n \leq 1$, false otherwise

The projected invariant P is in this case is therefore

P: $0 \leq a^2 \wedge n < (a+1)^2$

The determinate D is

D: $a^2 \leq n$

Following similar arguments to those used for the first solution, we get:

do $a^2 > n \longrightarrow a := a-1$ od.

Example 3.2 Quotient Remainder Problem

Given integers x and d, establish the quotient q, and remainder r, resulting from division of x by d without using the division operator. Stated more formally, we can express the postcondition R as

R: $(x = q * d + r) \wedge (0 \leq r < d)$

Also given is that $x \geq \phi$ and $d > \phi$. The postcondition can be rewritten as:

$$R: (x = q * d + r) \wedge \phi \leq r \wedge r < d$$

The free variables associated with the postcondition are r and q. The ranges for these two variables are:

$$q: (\phi \leq q \leq x) \quad \textit{note } r = \phi \textit{ and } d = 1 \textit{ gives } q = x$$
$$r: (\phi \leq r \leq x)$$

There are four possible combinations of projection values for the two variables. *Choosing r = x and q = ϕ as the projection values avoids any special compensation for* d. Substituting these values for r and q into the postcondition R yields:

$x = q * d + r \quad \Rightarrow$ true

$\phi \leq r \quad \Rightarrow$ *true*

$r < d \quad \Rightarrow$ *true for* $x < d$, *false for* $x \geq d$.

The projected invariant P is in this case therefore:

$$P: (x = q * d + r) \wedge (\phi \leq r)$$

The determinate D is

$$D: r < d$$

The determinate will be true directly provided x is initially less than D. However, the problem statement does not say anything about the relationship between the magnitude of x and of d and so our algorithm must be able to accommodate $x \geq d$.

Examining the invariant P we see that a suitable bound function follows directly from *the conjunct* $(\phi \leq r)$. *That is we can use*

$$t: r \geq \phi$$

The task that remains is to construct a mechanism that will establish the truth of the determinate D while maintaining the truth of the invariant P. Because division is not allowed (from the problem definition) an iterative process is needed. The projection values for r and q can be used to initialize the variables associated with the loop. The role of the loop body that we must construct in this instance is to establish the truth of determinate by decreasing the bound function while keeping P true. This will involve decreasing r. Reference to the invariant indicates that the only way to keep it true while decreasing r, is to reduce r by multiples of d. For each change in r by a multiple of d there will need to be an integral change in q to keep P true after each iteration. As our tentative loop structure we have at this stage:

```
r := x, q := ϕ;
do ? ⟶
        r := r - d, q := q + 1
od
```

Our next task is to derive a suitable guard for the loop. Examining the loop body we see that the maximum iterative capacity $\Psi_{max} = d$. We therefore have:

t: $r \geq \phi$
Ψmax: d
$t \geq \Psi_{max}$
$r \geq d$

We can now write down the full details of the algorithm. For the implementation of the various algorithms that will be presented we have used essentially Dijkstra's mini-language [3] with some extensions and variations that we have felt appropriate. Any deviations from Dijkstra's language will be identified as they are encountered.

The quotient remainder implementation is as given below:

```
lp φ ≤ x ^ φ < d ⟶
$vr r,q : integer ⟶
        r := x, q := φ;
$do r ≥ d ⟶
        r := r-d, q := q+1
$od;
$r: r < d => r, q
pl
```

Some comments on the conventions used in this implementation are in order. The construct:

```
lp B_R ⟶
pl
```

parenthesizes a mechanism whose role is to establish a defined postcondition R. It is *not* an iterative construct. It is like an if-construct but is more powerful because of its information hiding capability. The postcondition guard B_R will only allow entry to the mechanism when it is true (i.e. in this case $\phi \leq x \wedge \phi < d$ must be true).

By convention the only variables returned from the lp ... pl construct are those specified in the $r: state. *Variables specified in $r: state will only be bound and returned to the external environment when the determinate guard B_D is true.* In general we have

$r: B_D => < variable list to be returned >

In the example B_D takes the form r < d. Should the mechanism terminate in a state where the determinate guard is not true then the variables listed to be returned to the external environment will remain unbound. This condition can be used to signal a situation where it *definitely cannot* be inferred that the mechanism has established the defined postcondition. The same situation will apply in cases where the postcondition guard B_R is initially false.

All variables internal to the mechanism are defined internally in the $vr state. The conventions used for Pascal for definition and typing are employed. The iterative states of the mechanism used to establish the *major* postcondition are parenthesised by a $do .. $od construct. *Minor* loops included within the mechanism use the conventional do .. od notation. Where there is no preference for order of execution of a group of statements these statements are separated by commas rather than semicolons provided there is no evaluation contention. (For example, the two statements r := r-d and q := q+1 can be safely executed concurrently.) Where there is a possibility of contention the concurrent assignment operator used by Dijkstra [3] is adopted.

Concurrency is used throughout the paper as a tool to ensure that the invariant is maintained at all times rather than just initially, and after each iteration of a loop. We consider this requirement to be a central principle of good loop design. Applicative

consider this requirement to be a central principle of good loop design. Applicative languages provide an alternative means for meetng this requirement.

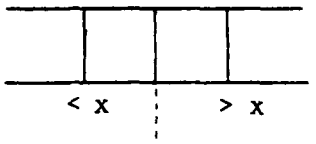
Other aspects of this variant of Dijkstra's mini-language will be defined as they are encountered in later problems.

Example 3.3 A Symmetric Binary Search

The binary search of an ordered array is one of the most widely cited examples in introductory computing science texts. Unfortunately solutions to the problem are rarely well formulated and well presented. We therefore felt at liberty to offer yet another somewhat different solution which hopefully stands up a little better to some of the criticisms we have hinted at.

The key to developing a "good" implementation for the binary search is in the formulation of a well formed postcondition in the first instance. Before we can do this we need a clear statement of the goal we are seeking. We are given an ordered array *a[1..nϕ] and a search value x and are required to establish whether or not a value* equal to x is present in the array. Our mechanism should return a boolean variable *found handle all cases including an empty array. It is possible that elements in the* array are not unique and x may be *less* than the first element or *greater* than the last element in the array. *Given these requirements we want a mechanism that maintains* a suitable invariant throughout.

The most natural requirements on how the mechanism should terminate are as follows. If x is *absent* from the array our mechanism should terminate having effectively accounted for all possible locations in the array. In this case we should be able to establish limits i, and n defining two array elements between which the value sought x, lies (apart from the two limiting cases). We might represent this schematically as follows:

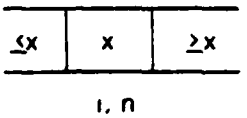


found = false

i = n-1

a[i] < x < a[n]

In the other case where x is present we again want to account for all elements in the array. The limits in this instance can most appropriately point to the array location where the value equal to x is located. Our algorithm, in this case, could terminate as our schematic diagram below indicates, i.e.



found = true

i = n

a[i] = x = a[n]

Taking into account the requirements for termination and using Dijkstra's advice about trying to formulate a postcondition that will permit generalization or weakening to an earlier state further from the end result, we are led to the following postcondition:

$$R: \quad (\phi \le i \le n\phi) \wedge \forall p\ ((1 \le p \le i) \Rightarrow (a[p] \le x)) \wedge (1 \le n \le n\phi+1) \wedge \forall q\ ((n \le q \le n\phi) \Rightarrow (a[q] \ge x)) \wedge ((i = n \wedge x = a[i] \wedge found) \vee (i+1 = n \wedge \neg found))$$

The free variables associated with the postcondition are i, n and found. Their ranges are:

i: $(\phi \le i \le n\phi)$
n: $(1 \le n \le n\phi+1$
found: (false, true)

We can choose $i = \phi$, $n := n\phi+1$ and found = false as the projection values. Substituting these values for i, n, and found into the postcondition yields:

$(\phi \le i \le n\phi) \wedge \forall p\ ((1 \le p \le i) \Rightarrow (a[p] \le x))$ $\Rightarrow$true
$(1 \le n \le n\phi+1) \wedge \forall q\ ((n \le q \le n\phi) \Rightarrow (a[q] \ge x))$ $\Rightarrow$true
$(i=n \wedge x=a[i] \wedge found) \vee (i+1=n \wedge \neg found)$ $\Rightarrow$ true for $n=\phi$, false for $n\phi > \phi$

The projected invariant P is in this case therefore:

P: $\phi \le i \wedge i \le n\phi \wedge 1 \le n \wedge n \le n\phi + 1$
$\wedge \forall p\ ((1 \le p \le i) \Rightarrow (a[p] \le x))$
$\wedge \forall q\ ((n \le q \le n\phi) \Rightarrow (a[q] \ge x))$

The determinate has the form:

D: $(i = n \wedge x = a[i] \wedge found) \vee (i+1 = n \wedge \neg found)$

The determinate is true directly for $n\phi = \phi$, i.e. for the empty array case. However, the general problem requires that the mechanism will also handle cases where $n\phi > \phi$. Note that when the projection values for *i* and *n* are substituted into the postcondition the antecedents of the two implications are both false, however, it follows from the definition of implication that an implication is true under these conditions.

Examining the invariant and the determinate we see that a suitable bound function is

t: $n - i \ge \phi$

The assignments $i = \phi$ and $n = n\phi + 1$ establish that in general x may occupy any of the positions in the range $i+1$ .. $n-1$ if it is present. Because the data set is ordered we are therefore at liberty to examine the average value of the range still to be considered i.e.

m := ((i+1) + (n−1)) div 2

which simplifies to

m := (i+n) div 2

There are clearly three possible outcomes for each location examined.

(i) **a[m] < x** : In this case we can safely assign i := m which will maintain the invariant, decrease the bound function, and hence make progress towards establishing the truth of the determinate.

(ii) **a[m] > x** : Here the assignment n := m can be safely made as it maintains the invariant and decreases the bound function.

(iii) **a[m] = x** : In this situation it is possible to make the assignments i := m and n := m. These assignments will maintain the invariant, decrease the bound function and presumably set up conditions for termination.

As all possibilities have been considered, we can now specify the initialization and the loop body.

As all possibilities have been considered, we can now specify the initialization and the loop body.

```
i := 0, n := n0+1;
do ? →
        m := (i+n) div 2;
        if a[m] < x → i := m
        [] a[m] > x → n := m
        [] a[m] = x → i := m, n := m
        fi
od
```

The task remaining is to derive a suitable loop guard. To do this we must determine the maximum iterative capacity for the smallest problem that requires iterative solution by the loop body.

The smallest general problem that requires iterative solution is that for $n_0 = 1$. In this case if the value sought x were present then the bound function would be reduced by 2, hence $\Psi_{max} = 2$. We therefore have:

t: $n-i \geq 0$
Ψ_{max}: 2
B: $t \geq \Psi_{max}$
B: $n-i \geq 2 \equiv i+1 < n$

The detailed implementation can now take the form:

```
Ip 0 ≤ n0 →
$vr i, m, n : integer; found:boolean →
        i := 0, n := n0+1;
$do i+1 < n →
        m := (i+n) div 2;
        if a[m] < x → i := m
        [] a[m] > x → n := m
        [] a[m] = x → i := m, n := m
        fi
$od;
$r: 0 ≤ n-i ≤ 1 => found := (i=n)
pI
```

In this example it is only possible to establish the truth of the determinate *after* the loop has terminated.

Several comments are in order about this binary search derivation and implementation. The postcondition on which the derivation of the algorithm is based is quite different from those usually used. It possesses a basic symmetry which is reflected in the final implementation. The use of quantified implications has avoided the need to talk in terms of $+\infty$ and $-\infty$ when dealing with x values that may be out of bounds with respect to the given array values (c.f. other derivations [4,7]). Furthermore, out-of-bounds cases do not require any special treatment. A number of other solutions have to treat the empty array (i.e. $n_0 = 0$) as a special case. The reason for this follows from the *degeneracy* of the postconditions that are usually employed. An often-used postcondition is built around:

$$\exists\, i : a[i] \leq x < a[i+1]$$

The problem with this is that at termination of the loop no distinction is made between $a[i] = x$ and $a[i] < x < a[i+1]$. Hence an additional test of the form:

found := (a[i] = x)

is needed. This has two disadvantages, it must be guarded to avoid an out-of-bounds array reference when an empty array is encountered, and secondly one more data element comparison is applied than is necessary to solve the problem (in fact the same comparison is applied twice).

The termination mechanism for the symmetric binary search does not allow termination in a degenerate state. A simpler test can therefore be applied to establish the presence or absence of x.

Another interesting attribute of the algorithm is that it stops as soon as the *first* value equal to x is encountered. It accomplishes this in a natural way without having to resort to the use of flags and an additional boolean test in the loop guard.

Removing the assignment for m and replacing it by i+1 leads directly to an ordered linear search algorithm for which the same postcondition is appropriate. Note that the variable m is not essential. It is also interesting to note that an analogous postcondition to that employed for the binary search could have been constructed for the square root problem discussed earlier.

No apology is made to those who may suggest that the present algorithm is "less efficient" than other alternatives when executed on a sequential computer.

4. On Loop Guards and Termination

In the preceding discussion on the binary search the remark was made about a simple change that could be used to convert the binary search algorithm into a linear ordered search algorithm satisfying the same postcondition. The interesting thing about the "converted" linear search is the way termination is brought about when x is present – that is by changing the least upper bound n.

There is, however, at stake here a much more fundamental issue which can be seen by considering a related problem, that of searching an unordered array a[1 .. n0] for some value x with the requirement that the algorithm should terminate as soon as x is found if it is present. Our mechanism should return a boolean variable *found* that is true if x is present and false otherwise. The mechanism should handle all cases including the empty array.

A suitable postcondition is:

$$R: \quad (0 \le i \le n0) \wedge \forall p(1 \le p \le i) \Rightarrow a[p] \neq x \wedge (i = n) \wedge ((n < n0 \wedge x = a[n+1] \wedge found) \vee (n = n0 \wedge \neg found)$$

The free variables associated with the postcondition are i, n and found. Their ranges are:

i: $(0 \le i \le n0)$
n: $(0 \le n \le n0)$
found: (false, true)

The role of i is as the greatest known lower bound on the segment of the array not containing x and the role of n is as the least *known* upper bound of the segment of the array not containing x. From the ranges for the free variables we can choose i = 0, n = n0, and found = false as the projection values. Substituting these values for i, n, and *found* into the postcondition yields:

$(0 \leq i \leq n0) \wedge \forall p\ (1 \leq p \leq i) \Rightarrow a[p] \neq x \quad \Rightarrow$ true

$(i = n) \wedge ((n < n0 \wedge x = a[n+1] \wedge found) \vee (n = n0 \wedge \neg found))$

$\Rightarrow$ true for $n0 = 0$, false for $n0 > 0$

The projected invariant P and determinate D are therefore as given below:

P: $(0 \leq i \leq n0) \wedge \forall p\ (1 \leq p \leq i) \Rightarrow a[p] \neq x$

D: $(i = n) \wedge ((n < n0 \wedge x = a[n+1] \wedge found) \vee (n = n0 \wedge \neg found)$.

Examining the invariant and the determinate we see that a suitable bound function is

t: $n - i \geq 0$

The projection values for i and n establish that if present, x may occupy any of the array positions in the range 1 to n. The task at hand is to examine the segment of the array where it is possible for x to be located. As the data is not ordered a linear search of the array by advancing i under the invariance of P is appropriate.

There are two possible outcomes for each location examined:

(i) **a[i+1] ≠ x** : is true in which case we can safely advance i by one under the invariance of P, while decreasing the bound function and hence making progress towards establishing the truth of D.

(ii) **a[i+1] = x** : In this situation it is not possible to advance i under the invariance of P. However, it is possible to decrease n, the least known upper bound for segment not containing x. This decrease in n, decreases the bound function and at the same time sets up conditions for terminating the loop.

As all possibilities have been considered we can now specify the initialization and the loop body:

```
i := 0, n := n0
do ? ⟶
        if a[i+1] ≠ x ⟶ i := i+1
        [] a[i+1] = x ⟶ n := i
        fi
od
```

Considering the smallest problem that requires iterative solution (i.e. $n0 = 1$) we discover that $\Psi_{max} = 1$. Therefore to derive the guard we have:

t: $n - i \geq 0$
Ψ_{max}: 1
B: $t \geq \Psi_{max}$
B: $n-i \geq 1 \Rightarrow \quad i < n$

The detailed implementation can therefore take the form:

```
|p 0 ≤ n0 →
$vr i,n:integer →
        i:= 0, n := n0;
$do i < n →
        if a[i+1] ≠ x → i := i+1
        [] a[i+1] = x → n := i
        fi
$od;
$r: i = n => found := n < n0
p|
```

We have deliberately chosen "i < n" as our guard in preference to "i ≠ n" as we are of the conviction that the role of the guard is to *protect* the mechanism - "i ≠ n" does not fulfill this role.

There are two ways in which the loop can terminate. The first involves "i" increasing stepwise until it reaches the value n. We refer to this mode of termination as *natural termination.* In the other case, when x is present, termination is brought about by reducing the upper bound n. We refer to this as *forced termination.* We reject the idea of using the assignment i := n as an alternative means of bringing *about forced termination because of its impact on the loop invariant.*

It is useful to introduce some additional terminology at this point. A loop that admits forced termination as well as natural termination is said to terminate in an *unresolved* state. Almost always it is necessary to *resolve* the termination state of such mechanisms. We would suggest as a principle of good programming style that this resolution should take place *after* the loop has terminated rather than in the loop body. This conforms to what we shall call the *law of separation of concerns* which states that *"any condition or mechanism that can obtain only in a single iteration of a loopshould be separated from the loop body".*

The advantage of consistently applying this principle is that it often cleans up and simplifies the structure of loop bodies. In addition, it gathers together mechanisms that naturally belong in the post-termination state. The idea of keeping loop bodies as simple as possible is important because they are conceptually more difficult to analyze and understand than non-iterative components.

There is also an important class of problems where the complementary situation applies, that is a component of the loop body apparently needs to be executed in all but the last iteration of a loop. A methodology for handling problems of this type is discussed in sections 5.3 and 5.4.

Returning to our solution to the linear search problem and its relationship to other solutions traditionally offered for this problem raises several important issues about programming style. The two traditional solutions we will give are based on the discussion by Feuer and Gehani [8]. (We have ignored solutions that employ sentinels for obvious reasons.)

The first solution is implemented in Pascal and the second in Dijkstra's mini-language:

(i)

```
found := false; i := 1;
while (i ≤ n) and (not found) do
        begin
                found := a[i] = x;
                i := i+1
        end;
if found then i := i-1
```

(ii)

```
i := 1;
do (i ≤ n) ∢ cand (a[i] ≠ x) ⟶ i := i+1 od;
found := i ≤ n
```

Considering solution (i) first, we see that the aim of the program seems to be to terminate with i = n if the value sought x is present in the array. This is only achieved as an afterthought by some variable patching once the loop has terminated. When we come to look for a meaningful invariant for this loop we run into a real headache. We might initially be hopeful that part of the invariant will be that a[1 .. i-1] does not contain x. But alas, nothing as straightforward as this is possible. The clumsiness of this solution is attributed to Pascal's lack of conditional logical operators (in this case cand). It is therefore suggested that the second implementation which uses a conditional *and* (i.e. cand) is a "better program" and so the use of conditional logical operators in guards is to be inferred as a good programming practice.

We would suggest that both solutions to the problem are built on poor formulations, and that they violate what we would consider to be several important principles of loop design. To uphold the spirit of structured programming we should strive to develop loops that have a *single* point of entry and exit. We would suggest that a guard consisting of a single condition conforms more closely to this ideal than one made up of the conjunction or disjunction of several conditions.

The use of conditional logical operators (which are not implemented in all languages (e.g. Pascal) has an unnecessary destructuring influence. For example, in the linear search (example (ii) in Dijkstra's mini-language) forcing the condition a[i] ≠ x onto the same level as the test i ≤ n destroys some of the natural structure of the solution because a[i] ≠ x can only be applied *after* it is established that the index test is true.

Another point about programming style that this example brings out relates to the storage of the n array elements in a[0..n-1] rather than the more conventional storage of the n elements in a[1..n]. For this (ii) would have the following form:

(ii')

```
i := 0
do i ≠ n cand (a[i] ≠ x) ⟶ i := i+1 od;
found := i < n
```

While in practice both formulations are perfectly valid, the former is an unnecessary complication which causes some confusion because of the way "i" is given its two roles. It must act firstly as a counter for the number of array elements processed and secondly as an array index. Using the convention a[0..n-1] the two roles are needlessly out of step by one. After one iteration it would seem more natural for the *first* element a[1] to have been processed *and* i to reflect a count of 1. Our implementation uses a[i+1] to accomplish this in a straightforward manner. The "+1" allows the mechanism to "peak-ahead" and then make the necessary adjustments to maintain the invariant. The other mechanism (ii') must also achieve lookahead. It does this by

having the index i of the array (after each iteration) always pointing at the *next* element to be processed rather than at the element that has just been processed. On termination "i" may point beyond the valid array indices (i.e. since a[n] is not defined). These characteristics make the a[0..n-1] scheme semantically clumsy.

Several other comments are in order, in the light of the proposed alternative methodology for constructing loops. One of the cornerstones of this methodology is the requirement that the guard for a loop should be intimately related to the bound function and to the iterative capacity. In contrast tests like a[i] ≠ x and (not found) used in (i) and (ii) bear no direct relationship to the bound function. Also, in the interests of proving termination, we would suggest that such tests should be excluded from loop guards.

Further weight is added to this suggestion if we re-examine how our original linear search algorithm was derived. Projection of the invariant from the postcondition clearly established that a[p] ≠ x belonged to the invariant, rather than the guard. Thankfully our methodology conforms to what might be considered as good programming practice.

Yet another way to think about this problem is to consider that guards should be defined for complete domains whereas conditions that have the potential to force early termination are subordinate and therefore belong in the body of the loop. Applying these ideas our integer square root algorithm can take a form analogous to the linear search i.e.

```
a := 0, n := n0;
do a < n →
        if (a+1)^2 ≤ n0 ⟹ a := a+1
        [] (a+1)^2 > n0 → n := a
        fi
od
```

As we shall see in many of the examples to be discussed the method of termination we have proposed has wide application. An explanation why it has not been widely used in the past is probably due to the influence of languages like Fortran which have required a fixed upper limit on loops.

The same principles can be applied to other data structures. For example, consider the three implementations in a Pascal-like language which search for some element x.

(i) Array

```
i := 0, n := n0;
do i ≠ n →
        if a[i+1] ≠ x → i := i+1
        [] a[i+1] = x → n := i
        fi
od;
found := n ≠ n0
```

(ii) Line

```
n@ := true;
i := eoln (input), n := n@;
do i ≠ n →
        if input ↑ ≠ x → get (input), i := eoln (input)
        [] input ↑ = x → n := i
        fi
od;
found := n ≠ n@
```

(iii) List

```
n@ := nil;
i := listhead, n := n@;
do i ≠ n →
        if i ↑.info ≠ x → i := i↑.link
        [] i↑.info = x → n := i
        fi
od;
found := n ≠ n@
```

In these examples we see how exactly the same control structure is used for all three implementations. There are obviously a number of advantages in using representation-independent control structures.

It is equally possible to apply the same techniques for file operations with languages like Cobol which do not have the same lookahead facilities as have been employed here.

From a methodological standpoint it makes a lot of sense, in appropriate applications, to develop solutions as if we were dealing with arrays of data. This makes it easier to define bound functions and prove termination. The type changes and the drawing of correspondences among such constructs as a[i+1], input↑ and i↑.info etc. is essentially a mechanical process. From a pedagogical viewpoint there are also advantages in that relatively detailed solutions to problems can be presented which are essentially transparent to the data representation.

Of course some will argue that we have made the list and file processing more complicated than need be by introducing more variables than are necessary to solve the problem. Before rejecting our proposal out-of-hand as just a programming trick, we would urge the reader to consider carefully the deeper implications that it may have for developing programs.

In the sections which follow we will frequently use and build on the techniques which we have described in this section. On a number of occasions we will deal with problems which are traditionally list or file oriented. In the light of the present discussion we will usually deal with these problems using arrays. However, they should be seen as general solutions with control structures which can conveniently accommodate other data representations.

5. Inductive Refinement and Loop Construction

"Beauty is the first test – there is no permanent place in computing
for ugly programs"

(with apologies to G.H. Hardy)

In the examples we have considered thus far the relationships among the bound functions, invariants and loop bodies have been clear-cut. As we move to slightly

more complicated problems it is useful to apply a more powerful constructive methodology. The guiding principle that is widely accepted by program designers for dealing with non-trivial problems is top-down design or stepwise refinement. This strategy is essentially a deductive one, in that the underlying constructive principle involves making transitions from the general to the specific.

As a method for breaking down problems into intellectually manageable components, top-down design is admirable. We would, however, suggest that for many problems, once this level of description has been reached, top-down design often fails to provide the constructive insight that we might hope for from such a design strategy. In such circumstances it is often more appropriate and more natural to apply what we shall describe as *inductive stepwise refinement.* The point in the program development process where it is most appropriate to apply inductive refinement is where we have a detailed and well-formed postcondition for an explicit task or sub-task. When inductive refinement is introduced under these conditions it can be directly related to details of the postcondition. At the same time it is often possible to effectively apply inductive refinement *without* having formally and explicitly defined the postcondition in terms of predicates etc.

The guiding constructive principle employed by inductive refinement is to start on the development of a solution to a problem by identification of the associated ii-mechanism. Repeating our earlier definition the ii-mechanism is the weakest iterative mechanism capable of establishing the postcondition. Having identified the ii-mechanism inductive stepwise refinement can then proceed by a sequence of generalizations until the mechanism has been refined to the stage where we nave an implementation for the general problem that we originally set out to solve.

At first sight it may appear that what we are advocating is a traditional bottom-up approach to program design. As succinctly pointed out by Turski [9], bottom-up design as traditionally employed is useful for creating software tools but is not generally regarded as a "problem-directed" programming methodology. In contrast to this, inductive refinement, because of its explicit relationship with the postcondition is a strongly problem-directed methodology. This relationship with the postcondition gives the method much more power and direction than traditional bottom-up design. We would see the role of inductive refinement as one of complementing the top-down development in the process of program design.

Polya in his classic works on Mathematical problem-solving [10] gives an excellent account of the importance and constructive role of induction (as distinct from mathematical induction) in the solution of problems. Just as the outcome of a mathematician's creative work is demonstrative reasoning, so, too, should be a finished program, for such reasoning provides a means of securing (as opposed to creating) knowledge in a science. The computing science literature clearly reflects this bias. Computing scientists should, however, not neglect knowledge concerned with the construction of programs. Efforts in this direction have unfortunately placed very little emphasis on the possible use of induction.

In general, to effectively apply induction in the solution of a problem the choice of an appropriate or relevant starting point is crucial. We would suggest that in employing induction in program development the ii-mechanism can often fulfill the role of an appropriate starting point. We will not continue to argue further in favour of the role of induction in program development. The reader is invited to consider the examples which follow and make his or her own conjectures about its value.

Before we can come to grips with the application of the inductive refinement methodology we must re-examine the underlying nature of the iterative process and then see the implications that this has for our present circumstances. The basic structure we are referring to is:

```
Initialization;
do B ⟶
  Loop-body
od
```

The role of the initialization is twofold. Firstly to provide a solution to the smallest problem (non-iterative) that the mechanism is designed to handle. Secondly, it must provide the setup for problems that require iterative solution.

When it is necessary to deal with more complex loop structures, where we have an iterative process (i.e. another loop) embedded inside a loop we need to re-examine what is the most appropriate initialization for both the inner and the outer loops, remembering that all loops require some form of initialization. Working to this basic loop design principle one fairly obvious structure for problems where we have a loop within a loop is:

```
Initialization (for outer loop);
do B0 ⟶
   Initialization (for inner loop);
   do B1 ⟶
      Loop-body
   od
od
```

What we need to recognize is that the outer loop has the obvious role of applying an iterative process (i.e. the inner loop and its initialization) *repeatedly*. We can regard the initialization for the outerloop as a *"Static initialization"* because it is only applied *once* whereas the initialization for the inner loop is a *"dynamic initialization"* in that it can be applied repeatedly. The relationship between the static and dynamic initializations can have an important influence on the composition of the loops. We will also see how this relationship can be used to help characterize and categorize loop structures into different classes.

5.1. Static Initialization

Probably the simplest nested loop structures are those that effectively require only a static initialization for the inner loop. We will now consider an example from this class because it gives a simple demonstration of how a ii-mechanism can be used to help solve a problem. The example also illustrates how the ideas for early termination of loops can be exploited.

Example 5.1 Searching a two-dimensional array

This problem involves searching a two dimensional array a[1 .. m0, 1 .. n0] for some value x. If x is present in the array the mechanism should return indices that can be used to identify the row and column where x was located. A boolean variable *found* should also be established and set to true if x is present, and set to false otherwise. The algorithm should terminate as soon as x is found if it is present. Given these requirements we may propose the following postcondition as suitable for use in developing the searching algorithm.

R: $(0 \le i \le m0) \wedge (0 \le j \le n0) \wedge \forall p,q$
$(1 \le p \le i) \wedge (1 \le q \le j) \Rightarrow a[p,q] \neq X \wedge (i = m) \wedge (j = n)$
$\wedge ((m < m0 \wedge x = a[m+1, n \bmod n0+1] \wedge found) \vee$
$(m = m0 \wedge \neg found))$

In the postcondition m is the greatest known lower bound for the *complete* rows not containing x and n is the least known upper bound for columns not containing x corresponding to an assigned row value.

The ranges associated with the variables in R are:

i: $(0 \le i \le m0)$ j: $(0 \le j \le n0)$ m: $(0 \le m \le m0)$ n: $(0 \le n \le n0)$

We can choose as projection values for these variables $i = 0$, $j = 0$, $m = m0$ and $n = n0$. Projecting the invariant P and determinate D from the postcondition R we get:

P: $(0 \le i \le m0) \wedge (0 \le j \le n0) \wedge \forall p,q$
$(1 \le p \le i) \wedge (1 \le q \le j) \Rightarrow a[p,q] \neq x$

D: $i = m \wedge j = n \wedge ((m < m0 \wedge x = a[m+1, n \bmod n0+1] \wedge found)$
$\vee (m = m0 \wedge \neg found)$

A possible bound function would be

t: $m - i + n - j \ge 0$

Our task now is to develop a loop that will decrease the bound function with each iteration. Applying the inductive refinement methodology we are looking for in the first instance, the *weakest* iterative mechanism capable of establishing the postcondition R. To make progress towards establishing R we have four options, decreasing m, and n or increasing i and j. The weakest possible mechanism is one that will only need to change *one* of these four variables. For example a mechanism that changed just the j index (except possibly on termination) and which could terminate on finding x would qualify as the ii-mechanism. It processes a single row of the array.

```
j:= 0, i:= 0;
do j < n =>
   if a[i+1,j+1] ≠ x ⟶ j:= j+1
   [] a[i+1,j+1] = x ⟶ n:= j
   fi
od
```

This loop can terminate in only one of two ways, with x found and the least known upper bound for the segment not containing x redefined, or with it established that x is not in the specified row.

The mechanism we have described is only sufficient to process a single row of a two dimensional array. To solve our original problem our mechanism must be generalized so that it can handle a two dimensional array with more than one row. Under these conditions, to make further process towards termination it will be necessary to change another variable i, (ie the row variable). Each row can be processed in essentially the same manner and so the generalization to the processing of m0 rows is straightforward. All that is needed is to apply the single row mechanism repeatedly until either x is found or it has been established that x is not present in all rows. At the point where it is established that x is present, using our original mechanism it is implicit that the least upper bound on the number of complete rows not containing x is redefined. This step must be included explicitly (ie by m := i) when our mechanism is generalized to handle more than one row. This assignment also has the effect of terminating the row searching as soon as x is found. We are therefore led to the following implementation.

```
Ip 0 ≤ m0 ^ 0 ≤ n0 →
$vr i,j,m,n:integer →
   i:=0, m:=m0;

$do i < m →
   j:=0, n := n0;

   do j < n →
      if a[i+1, j+1] ≠ x → j:= j+1
      [] a[i+1,j+1] = x → n:= j, m:= i
      fi
   od;
   i:= i+1
$od
$r:  m ≤ m0 ^ n ≤ n0 => found := m < m0
```

In handling termination for this problem we have applied the principles that were outlined in the previous section for cleanly terminating loops.

It is interesting to compare the above solution to the problem with the solution given by Gries ([4] pp 183–184). Our mechanism has the same underlying structure as the mechanism we used to solve the corresponding one dimensional problem. This is somewhat reassuring. Gries' algorithm uses only a *single* loop to solve the problem. It could be argued that this amounts to a superior simplification of the problem over our nested-loop solution. On the other hand it can be argued with equally as much weight that the problem is two-dimensional and therefore naturally supports a nested- loop implementation. The guard used by Gries has the form

B: i ≠ m cand x ≠ a[i,j]

In contrast, our method of projecting the invariant from the post condition naturally relegates the test involving x to the loop body. Gries' implementation relies explicitly on the use of the *cand* operator in the guard. Hence his solution is probably more language dependent than is desirable. We would suggest that the logical components of a guard should, if at all possible, have a direct relationship with the bound function. x ≠ a[i, j], certainly does not have this property.

In summary, this problem which in essence is relatively simple, causes most adherents of structured programming considerable discomfort particularly if they are unwilling to resort to either cands, breaks or gotos. We would suggest that our solution conforms naturally to structured programming principles.

5.2. Inductive Initialization

We will now move onto a set of examples that employ a simple form of dynamic initialization for the inner loop.

Example 5.2 The Inventory Report Problem

The inventory report problem is a well-known programming problem. In presenting the problem we have stripped away most of the extraneous detail so that we can concentrate on the methodological issues that the problem raises. Stated in its simplified form we are given two arrays of length n0. The first array is a product-identifier array which contains a set of incoming and outgoing transactions for various products, ordered by their product identifier. The elements in the second array form a one-to-one correspondence with the elements in the first array. The elements stored in this array are positive or negative quantities each representing a particular transaction for the corresponding product in the first array.

Schematically the data may be represented as:

product g1 g1 g1 g2 g2g3g3....

transaction −7 +2 +6 +2 +9−9−6....

The task at hand is, given the information about transactions, to produce a report of the net quantity of each product currently held in stock.

Sets can be used to give the simplest description of the postcondition. We have

$$G = \bigcup_{}^{k} Gi \quad \text{with } Gi \cap Gj = \phi \text{ for all } i \neq j$$
$$Ti = \sum_{g \in Gi} tg$$

Here G represents the union of all products and tg is a transaction for product Gi. With these definitions our postcondition takes the form

$$R: (0 \leq i \leq k) \wedge \forall p\ (1 \leq p \leq i) \Rightarrow (T_p = \sum_{g \in Gi} tg \qquad \wedge (i = k)$$

Our formulation of the problem has hidden the explicit count of the total number of transactions nϕ that need to be processed. This will obviously need to be accommodated in our implementation. Projecting the invariant and the determinate from the postcondition using i = ϕ we get:

$$P: (\phi \leq i) \wedge (i \leq k) \wedge \forall p\ (1 \leq p \leq i) \Rightarrow T_p = \sum_{g \in Gp} tg$$

D: i = k

In general the data may consist of one or more products and for each product there may be one or more transactions. In the first instance we are looking for the weakest iterative mechanism capable of establishing the postcondition. It is possible that we may have to deal with *only one* product for which there may be more than one transaction. A mechanism to solve this problem would correspond to the π-mechanism for the general problem.

In formulating the π−mechanism to total the transactions for a *particular* product it is implicit that all the transactions are for that product. Therefore to be strictly correct in formulating our π−mechanism, we should include a check to ensure that transactions for only one product are being processed. Termination should take place if it is found that there are transactions for other than the particular product being processed. Our π−mechanism can therefore take the form

```
j := 1, n := nϕ;
T := t[j];

do j < n ⟶
   if g[j] = g[j+1] ⟶ j, T := j+1, T+t[j+1]
   [] g[j] ≠ g[j+1] ⟶ n := j
   fi
od;
write (g[j], T)
```

Notice that this loop will total all transactions for a single product. If another product is encountered the algorithm immediately terminates.

As was stated earlier for the general case we will need to be able to handle more than one product. By applying the process that will handle a single product iteratively over all products, we will achieve the generalization necessary to solve our original problem. This generalization is relatively straightforward. With each iteration our refined mechanism we will require that another product has been processed.

Our final algorithm can therefore take the form:

```
|p ⓪ ≤ n⓪ ⟶
$vr j,n,T: integer ⟶
   j := ⓪;

$do j < n⓪ ⟶
   n := n⓪;
   j, T := j+1, t[j+1];

   do j < n ⟶
      if g[j] = g[j+1] ⟶ j, T := j+1, T+t[j+1]
      [] g[j] ≠ g[j+1] ⟶ n := j
   od;
   write (g[j],T)
$od;
$r: j = n⓪ => true
|p
```

In making the generalization to deal with more than one product the only change necessary was to ensure that the initialization for the product-processing loop was made dynamic by introducing the assignment j := j+1 in place of j := 1. Solutions to this problem have been discussed at length in a recent review of programming methodologies given by Bergland [11]. Essentially the same problem has also been discussed by Jackson [5]. Bergland, who describes the problem as the "McDonald's problem" gives two solutions of interest, one that was derived using Jackson's methodology and a second solution that was derived using Dijkstra's methodology. The Jackson-style solution which reads the data from a file with a Cobol-type read has the following form:

```
read (g2,t);
do while (not eof)
   T := ⓪;
   g1 := g2;
   do while (g1 = g2) and (not eof)
      T := T+t;
      read (g2,t)
   od;
   write (g1,T)
od
```

Our present algorithm is superficially quite close to Jackson's and Bergland's solutions, however the difference which exists raises what we consider are several important methodological issues of loop construction. The difference occurs in the way the initialization is handled for the loop that does the processing of all the transactions for a given product. *We take the view stated earlier that the initialization for a loop should account for the smallest defined problem that the loop is required to solve* - in this case that of dealing with a single transaction for a particular product. Neither the Jackson nor the Bergland solutions conform to this ideal for initialization. As a result their loops are always forced to execute at least once with the consequence that the tests (g1 = g2) and (not eof) are executed needlessly once for each iteration of the outer loop.

A second fundamental issue that the differences in implementation raises relates to the progress that the outer loop is seen to make with each iteration. This progress for Jackson and Bergland's solutions (ie the reading of the next transaction from the file) is *hidden* within the inner loop. We consider this to be a poor design principle as

it makes progress towards termination of the outer loop totally dependent on the inner loop. This makes proving termination of the outer loop more complicated than is necessary.

It should be stressed that the issues that we have raised with this problem are not artefacts of the fact that we have worked with arrays rather than files that require either Pascal-like or Cobol-like read operations. Our implementation can be converted to either of the file solutions by straightforward application of the techniques described in the preceding section.

Bergland [11] suggests that this problem illustrates two important advantages of Jackson's methodology. The first advantage is that the methodology yields a solution "that seems to model the problem best". The need to iterate over all transactions for a product and then over all products would seem to naturally suggest a structure consisting of two loops. While we are in agreement with the observation about the natural structure of the problem we would suggest that the solution obtained by inductive refinement arrives at the same goal for a deeper reason. What we are claiming is that the inductive refinement methodology should, if correctly applied, yield solutions that naturally follow the inherent data structure in the problem. This should follow because the underpinning constructive principle being applied always attempts to establish the postcondition by changing the least number of variables. Subsequent generalizations required also apply the same principle. We will attempt to demonstrate in subsequent examples that the scope for application of these principles appears to be considerably wider than a methodology that is tied to data structuring.

The second advantage of Jackson's methodology noted by Bergland is that "people are more likely to derive the preferred solution using data structure design than by any of the other methods". We are not prepared to argue one way or another whether this is true. We would, however, suggest from somewhat limited experience, that inductive refinement has at least as much potential as Jackson's methodology for consistent application. This follows from the fact that the weakest iterative mechanism for establishing the postcondition is usually well - defined. Subsequent generalizations should also follow consistently.

As a second point of reference it is worthwhile making a few brief comments about the solution quoted by Bergland that was derived using Dijkstra's methodology. This solution uses a single loop to solve the problem and consequently does not make explicit the two modes of iteration i.e. iteration over transactions for a product, and iterations over products. It has the form:

```
m1, i, c1 := 1, 0, 0;
do m1 ≠ m →
   if f[m1] = f[m1+1] → c1 := c1+q[m1]
   [] f[m1] ≠ f[m1+1] → i := i+1; c[i] := c1+q[m1], c1 := 0
   fi;
   m1 := m1+1
od
```

In one sense this solution is "simpler" than any of the previous solutions but we would suggest that the simplicity has been gained only at the expense of a more complicated invariant, introduction of a sentinel, and a destruction of the natural structure of the problem.

We would suggest that this mechanism "lags one step behind" which makes it more difficult to characterize c1 with an invariant properly. Our other criticism is that the algorithm relies on the use of a sentinel to handle termination, a practice which we would consider should be avoided.

As a final contrast between inductive refinement and Dijkstra's methodology we would suggest that the present example illustrates that inductive refinement is not necessarily so heavily dependent on having a precise mathematical formulation of the postcondition. This suggestion is not intended to imply that it is unnecessary to establish a precise mathematical formulation of the postcondition. To the contrary, we would claim that the more precisely the postcondition is formulated the more explicit is the guidance that inductive refinement can give in the constructive development of algorithms.

Example 5.3 Prime Factorization

The problem of prime factorization illustrates several useful points about loop construction. The problem has been discussed in some detail by Alagic and Arbib [12]. In discussing the problem we will attempt to formulate the algorithm so that it meets their requirements as we subsequently wish to make a comparison with their implementation. This involves explicitly saving an array of all factors including multiple occurrences. It has been assumed that a source of all ordered primes is available in an array. In our solution we will assume that there is a function available, called nextprime, which returns the next prime when given the current prime. This function is to be initialized by giving it the starting value 1. A precisely formulated postcondition for this problem is rather unwieldy. We will give a somewhat simplified version of R. Given the integer z_0 to be factored, the ordered primes P_1, P_2, ... P_x, and the corresponding exponents e_1, e_2, ... e_k which are all greater than or equal to zero, we can write our postcondition as

$$R:\ (1 \le z \le z_0) \wedge z.p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k} = z_0$$

$$\wedge \forall q\ (1 \le q \le i) \quad z \bmod p_q \neq 0 \wedge z=1 \wedge i=k$$

At this stage we could go ahead and project out the invariant and determinant and proceed as we have done previously. Instead we will attempt to show how inductive refinement can be applied to develop the algorithm without having to rely on a precise formulation of the postcondition.

If we are to proceed along these lines we must first ask the question, what is the weakest iterative mechanism that could solve the problem? This would obviously correspond to the situation where the integer z_0 is equal to a single prime (in the simplest case the first prime) raised to some power. To solve this problem we will need to reduce the number to be factored by repeated division by p until p will no longer divide into the quotient. Our ii-mechanism can therefore take the form:

```
z := z0, p := 1, i := 0;
p := nextprime(p);

do z mod p = 0 -->
   i, z, f[i+1] := i+1, z div p, p
od
```

If our mechanism is to handle the case where z_0 may contain more than one prime factor we will need to generalize the mechanism so that other prime factors can be tried. To do this, all we will need to do is apply our ii-mechanism iteratively. This brings us to the problem of deciding under what conditions should our more general mechanism terminate. Obviously our mechanism can only be applied while the primes being considered as factors are less than or equal to z.

We can therefore propose as our prime factorization algorithm the following implementation:

```
z := z0, p := 1, i := 0
do z ≥' p →
   p := nextprime(p);

   do z mod p = 0 →
      i, z, f[i+1] := i+1, z div p, p
   od
od
```

At this point we might be content with the solution we have obtained and not pursue the development further. Instead we will take the opportunity to consider two ways of increasing the efficiency of the loops in an existing algorithm.

One way to improve the efficiency of an existing loop is to look for a way to change the guard of the loop so that fewer iterations are needed to achieve the same goal. A knowledge of number theory, common sense, or a careful analysis of the problem would tell us that when the quotient (ie z div p) is less than the current prime p there can be no other prime factors. Applying this criterion will allow us to stop the outer loop earlier than is possible with the test $z \geq' p$. This refinement will be included in our final implementation of the algorithm.

A second way to improve the efficiency of an existing loop is to increase its average iterative strength. This will have the consequence of on average decreasing the bound function by larger amounts with each iteration. The iterative strength of the outer loop in our example is tied to successive primes and there does not appear to be a way of eliminating any of the primes from consideration. Therefore our second strategy is not applicable to the outer loop. However this idea can be applied to the inner loop. To understand this let us focus back on our ii-mechanism where if the original z_0 was a prime raised to a large power (eg $z_0 = 2^{23}$) we could use the standard binary doubling strategy for exponentiation which is logarithmic in the exponent. We have not bothered to include the second refinement in our final algorithm. We will however in other examples use this second strategy for improving the efficiency of existing loops. Our final algorithm takes the following form.

```
|p 1 < z0 →
$vr z,p,i:integer →
   z := z0, p := 1, i := 0;
$rp
   p := nextprime (p);
   do z mod p = 0 →
      i, f[i+1], z := i+1, p, z div p
   od;
$pr → z div p < p;
$pt z > 1 → i, f[i+1] := i+1, z;
$r: z div p < p => f, i
p|
```

Several observations can be made about this implementation. Firstly a repeat ...until loop has been used in preference to a do-loop because termination could not be possible until at least one prime has been tried as a factor and so it is most appropriate to test the guard at the *end* of the loop. Application of the guard at the beginning of the loop in this case requires some unnecessary duplication of code. As is usually the case the ii-mechanism can terminate in one of two states. These termination states must be accommodated in our implementation.

The implementation in Pascal of a prime factorization algorithm described by Alagic and Arbib [8] is given below.

```
begin
   t := 0; k := 0;
   q := n div d[0]; r := n mod d[0];
   while (r = 0) or q > d[k] do
   begin
      if r = 0 then
      begin
         t := t+1; f[t] := d[k];
         n := q
      end
      else
         k := k+1;
      q := n div d[k];
      r := n mod d[k]
      end;
   if n > 1 then
      begin
      t := t+1;
      f[t] := n
      end
end
```

In comparing the two implementations we see that by using a $\tilde{\Pi}$-mechanism in preference to an alternative construct we have avoided the need to apply the factor test (ie *r = 0) twice in succession. We would also suggest that the first solution is simpler and is* the "preferred solution" because there is a loop over all possible prime factors and a second loop overmultiple occurrences of a given prime factor.

As a final observation about the development of the solution using inductive *refinement although the author was aware of Alagic and Arbib's solution at the time,* the reasoning described here is a fairly faithful transcription of how the final solution was arrived at. We would claim that inductive refinement gave strong constructive guidance and turned this problem into a relatively trivial one. The logical development followed by Alagic and Arbib does not appear to be quite so straightforward at least to this author.

5.3. Deductive Initialization

In our earlier discussion of the structure of loops we made the observation that *for nested loops the role of the outer loop is that of applying an iterative process and* its accompanying initialization steps repeatedly. We have in the preceding two subsections examined cases where the iterative processes that have been applied repeatedly were treated uniformly in that no special requirements were needed for the first *application of the iterative process. There is however an important class of problems* where the requirements for the *first* application of an iterative process differ from the requirements of subsequent applications. For this class of problems the first application of the iterative process (or $\tilde{\Pi}$-mechanism) needs to assume the role of an initializing step for subsequent applications of a loop-body that incorporates the $\tilde{\Pi}$-mechanism. The dynamic initialization required for the $\tilde{\Pi}$-mechanism is referred to as a *Δ-mechanism.* Its structure must be deduced from the state of the computation when the $\tilde{\Pi}$-mechanism terminates without having established the postcondition. The role of the Δ-mechanism is to change the state of the computation to a configuration *where, if the postcondition still has not been established, it is possible to again apply* the $\tilde{\Pi}$-mechanism. The underlying structure of the loops for this class of problems is:

```
Π-mechanism: (initializing step)
do B_I →
   Δ-mechanism;
   Π-mechanism
od
```

Traditionally problems discussed in the literature that have these initialization requirements have loop bodies that effectively apply the Π-mechanism *first* ie

```
do B_I →
   Π-mechanism;
   if ¬B_B → Δ-mechanism
   [] B_I → skip
   fi
od
```

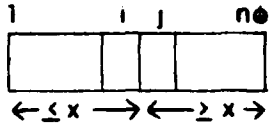
Because it is possible for the Π-mechanism to establish the postcondition it is essential to protect the Δ-mechanism with an extra guard. This amounts to loop overdesign. We would suggest that this latter implementation is inferior because it applies unnecessary tests and also because it violates the basic idea that each loop should have an appropriate initializing step.

At this point we are ready to consider several classic examples that belong to this important class of problems.

Example 5.4 Partitioning an Array

The problem of partitioning an integer array a[1..n⊕] about some value x is a familiar problem in computing science. As we have mentioned it is a prototype for a class of problems that are amenable to a similar method of solution.

Stating the partitioning problem a little more precisely the task at hand is to partition the array elements about x such that all elements less than or equal to x are in one partition and all those elements greater than or equal to x are in the other partition. There are a number of variations on this basic problem which depend on the origin of the partitioning value x and whether or not it is present in the array. We will concentrate on the general formulation of the problem where no assumptions are *made about x. The requirements given suggest the following schematic diagram as a* possible terminating condition.



Where all a[1..i] are less than or equal to x, and all a[j..n⊕] are greater than or equal to x. Of course it is possible that x may be either greater than the largest element in the array or less than the smallest element in the array. It is desirable that *our formulation handle these special cases smoothly and naturally. Given these* requirements we may propose the following postcondition as suitable for use in developing the partitioning algorithm.

$$R:\ (\oplus \le i \le n\oplus) \wedge \forall p\ ((1 \le p \le i) \Rightarrow (a[p] \le x)) \wedge (1 \le j \le n\oplus+1) \wedge \forall q\ ((j \le q \le n\oplus) \Rightarrow (a[q] \ge x)) \wedge i = j-1$$

It is interesting to note that most postconditions given in the literature for this problem are formulated in such a way that i and j are allowed to "crossover" [2.4.12].

Crossover seems to be an artefact of existing implementations rather than something that is inherent in the problem.

Having formulated the postcondition we can now proceed with the development of the algorithm. The free variables associated with R are i and j. The ranges for these variables are already supplied. We have:

$$i: \quad (\phi \leq i \leq n\phi)$$
$$j: \quad (1 \leq j \leq n\phi+1)$$

From these ranges we are free to choose as our projection values $i = \phi$ and $j = n\phi+1$. Substituting these values for i and j into the postcondition R yields the following invariant P and determinate D:

$$P: \quad (\phi \leq i \leq n\phi) \wedge \forall p\ (1 \leq p \leq i) \Rightarrow (a[p] \leq x))$$
$$\wedge\ (1 \leq j \leq n\phi+1) \wedge \forall q\ (j \leq q \leq n\phi) \Rightarrow$$
$$(a[q] \geq x))$$

$$D: \quad i = j-1$$

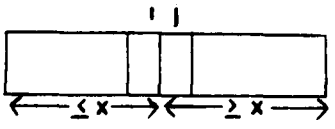
Because the antecedents of the two implications are false for the chosen projection values the two implications will still be true. The determinate is only true at projection for $n\phi = \phi$ and false for larger values of $n\phi$.

In this case the bound function can be taken directly from the determinate i.e.

$$t: \quad j - i - 1 \geq \phi$$

We might at this stage be tempted to suggest that a suitable guard would be either $i < j-1$ or $i \neq j-1$. As we will see a little later this suggestion is premature because we have not taken into account the iterative capacity of the loop.

Our task now is to develop the loop body. Applying the inductive refinement methodology we are looking in the first instance for the weakest general iterative mechanism capable of establishing the postcondition R. To establish R we have three options: increasing i, decreasing j, and/or changing the configuration of the elements in the array. If the array were already in a partitioned configuration then the problem would reduce to one of establishing the appropriate values for i and j that satisfied the determinate. A mechanism capable of establishing the determinate under these conditions can be considered as the Π-mechanism for the problem. The weakest possible mechanism would be one that changed either i or j but not both. However, the more general mechanism is one capable of establishing R for *all* possible final i and j values. That is, it would be possible for some data configurations to establish R by changing just two variables, i and j, without moving any elements in the array. Developing the Π-mechanism in itself can be seen as a separate problem which could be solved by applying similar methods. Instead, having identified the Π-mechanism, we will go straight to its implementation as it is very straightforward. In essence all we need is a mechanism to *confirm* the following configuration for all allowable i, and j.



In trying to confirm R, the mechanism we propose must maintain the invariant P. Should, however, the Π-mechanism enter a state where it is not possible to make further progress without changing other variables (ie changing the array configuration) the Π-mechanism must terminate and some other action must be taken. Our Π-mechanism to handle all possible partitioned configurations can take the form

$\bar{\Pi}$-mechanism:

```
il := j-1;
do i < il →
   if a[i+1] ≤ x → i := i+1
   [] a[i+1] > x → il := i
   fi
od;
jl := i+1;
do jl < j →
   if a[j-1] ≥ x → j := j-1
   [] a[j-1] < x → jl := j
   fi
od
```

Whenever the $\bar{\Pi}$-mechanism terminates it can be in one of two states, either R will have been confirmed or the computation will be in what we shall call a *Δ-state* and R will still not have been established. A Δ-state is characterized by the conjunction of the conditions that bring about early termination of the $\bar{\Pi}$-mechanism i.e. in this case

$$\Delta\text{-state: } a[i+1] > x \wedge a[j-1] < x$$

This state reflects the detection of two wrongly partitioned elements. To make progress under the invariance of P the complements of *both* the conjuncts in the Δ-state must be established. That is, we require

$$/\ a[i+1] \leq x \wedge a[j-1] \geq x$$

A swap of the elements a[i+1] and a[j-1] will achieve this. It will also set up a condition that will allow i to increase by one, and j to decrease by one under the invariance of P. The Δ-mechanism can therefore take the form:

Δ-mechanism:

```
i, j, a[i+1], a[j-1] := i+1, j-1, a[j-1], a[i+1]
```

After application of the Δ-mechanism the computation will again be in a state where it is possible to apply the $\bar{\Pi}$-mechanism in an effort to confirm the postcondition R. The loop to establish the determinate will therefore consist of alternatively applying the $\bar{\Pi}$-mechanism and the Δ-mechanism.

The seemingly obvious way to combine these two components of the loop body is as follows.

```
do B →
   Π̄-mechanism;
   Δ-mechanism
od
```

Unfortunately this method of composition for the two mechanisms turns out to be unsatisfactory in this case for the fundamental reason explained in the introduction to this subsection. We are therefore led to the idea of applying the $\bar{\Pi}$-mechanism at initialization and then constructing a loop body with the Δ-mechanism at initialization and then constructing a loop body with the Δ-mechanism preceding the $\bar{\Pi}$-mechanism.

The only task that remains is to determine the guard for the loop. To do this we must first determine the iterative capacity of the loop body. The loop body has the

following structure:

```
do ? →
   i, j, a[i+1], a[j-1] := i+1, j-1, a[j-1], a[i+1];
   ii-mechanism
od
```

The smallest general problem requiring *iterative* solution by the loop body will be a problem where n⊙ = 2 and a[i+1] > x and a[j-1] < x. For this problem the Π-mechanism applied externally will make no progress, the loop will then be entered, and the swap will be made together with the accompanying changes to i and j. The Π-mechanism will then again be encountered but will make no further progress as R will have been established. The maximum iterative capacity of the loop is only 2. It is derived from the two statements i := i+1 and j := j-1 in the Δ-mechanism which decrease the bound function by 2. We can now proceed with the determination of the guard noting that the bound function was determined earlier.

```
t:  j-i-1 ≥ ⊙
Ψmax:   2
B:  t ≥ Ψmax
B:  j-i-1 ≥ 2 ≡ i+3 ≤ j ≡ i < j-2
```

The full implementation for the partitioning mechanism can now be described.

```
ip ⊙ ≤ n⊙ →
$vr i, j, ii, ji:integer →
   i := ⊙, j := n⊙+1;

$do i < j-2 →
   i, j, a[i+1], a[j-1] := i+1, j-1, a[j-1], a[i+1];
$i: ii := j-1;
   do i < ii →
      if a[i+1] ≤ x → i := i+1
      [] a[i+1] > x → ii := i
      fi
   od;
   ji := i+1;
   do ji < j →
      if a[j-1] ≥ x → j := j-1
      [] a[j-1] < x → ji := j
      fi
   od
$od;
$r: (i = j-1) => i, j, a
pi
```

Several comments can be made about this mechanism. Referring back to our original structure for nested loops we see that the Π-mechanism occurs in the algorithm text in two places, first as an initializing step for the outer loop, and secondly as part of the body of the outer loop. We could have gone right ahead and written in the Π-mechanism text in the two places where it is needed. Instead we have chosen to use the following $i: state convention:

This convention is taken to mean that loops with a defined $i: state begin execution at that point in the program text. There are a number of other ways we could have resolved this problem including the use of the structure employed by Knuth that possesses a middle exit [15]. We could also have implemented the ii-mechanism as

a procedure and made two procedure calls, one at initialization and one in the loop body. Unfortunately language designers have failed to recognize this fundamental control structure and the problem of needing to write the text of the ĩi-mechanism twice. Wirth [16] discusses such problems as loops with "an exit in the middle", and then goes on to consider two examples which he considers demonstrate that there is no "real necessity" for such loop structures. We would suggest to the contrary as our partitioning example demonstrates there are many problems that can best be solved using the loop construction we have suggested.

It is interesting to compare the present implementation with that for a variation on the same problem that was implemented by applying Dijkstra's methodology [4]. The important different lies in the fact that the ĩi-mechanism is iterative rather than consisting of a set of components of an alternative construct, i.e.

```
if  a[i] ≤ x ⟶ i := i+1
[]  a[j] > x ⟶ j := j-1
[]  a[j] ≤ x < a[i] ⟶ swap (a[i], b[j]); i := i+1, j := j-1
fi
```

For the most general problem the difference in structure makes very little difference, however if x is present in the array (as for the partitioning mechanism in quicksort [2]) then an iterative ĩi-mechanism is more efficient because its greater iterative strength will, on average, reduce the number of times the loop guard is applied.

We would suggest that the preferred solution for partitioning involves an iterative ĩi-mechanism rather than a alternative construct. The modifications to our partitioning mechanism most appropriate for application with quicksort are discussed elsewhere [13].

As a point of programming style it is interesting to note the difference between the present solutions and the Dijkstra-style solutions for the partitioning problem and the inventory report problem. In both instances our methodology has led to iterative ĩi-mechanisms whereas the Dijkstra-style solutions have employed alternative constructs.

Example 5.5 A Text Formatting Problem

The text formatting problem that we wish to consider involves input of data consisting of one or more words separated by one or more spaces. No spaces precede the first word but there may be spaces trailing the last word. We will assume for simplicity that this data is stored as an array of n characters. The requirements for the formatted output are that successive words should be separated by a *single* space, and there should be no leading or trailing spaces. Essentially the same problem is discussed in detail by Dijkstra [14]. We have, however, generalized it so that the data does not contain a sentinel. The other issues raised by Dijkstra's specification of the problem are peripheral.

To develop a solution to this problem using inductive refinement in the first instance we are looking for the weakest general iterative mechanism capable of establishing the postcondition (i.e. the output format specified). Assuming there are neither leading blanks nor is there an array of only blanks then we might propose as the weakest iterative mechanism a procedure that "reads" a *single* word and writes it out. For this purpose we may propose the following mechanism. (We have chosen not to specify the procedure for writing out a single word explicitly.)

```
n := n@, j := @, i := @, space := ' ';
do i < n ⟶
   if a[i+1] ≠ space ⟶ i, j, w[j+1] := i+1, j+1, a[i+1]
   [] a[i+1] = space ⟶ n := i
   fi
od;
writeword (w,j)
```

This mechanism will certainly handle a single word, but referring back to our specifications, we see that the smallest general problem may include zero or more trailing spaces. Our more general ii-mechanism should accommodate trailing spaces. The following steps will therefore need to be added to the mechanism described above.

```
n := n@;
do i < n ⟶
   if a[i+1] = space ⟶ i := i+1
   [] a[i+1] ≠ space ⟶ n := i
   fi
od
```

We now have a mechanism to handle a single word in the general case.

Our next step is to look for a way to generalize it so that it can handle more than one word. The important thing in dealing with multiple words is that they all must be separated by single spaces. It follows that we must incorporate the "writing of a single space" into our mechanism. Clearly the first word can be written unconditionally. *All* remaining words can only be written after a space has been written. It follows that the ii-mechanism should be applied as the initializing step. The basic loop structure will therefore be

```
ii-mechanism;
do i < n ⟶
   write (space);

   ii-mechanism
od
```

The detailed implementation is as follows:

```
|p 0 ≤ n0 →
$c:space:char → space := ' ';
$vri,j,n,w[1..n0]:integer →
   i := 0;


$do i < n0 →
   write (space);

$i: n := n0, j :=0;

   do i < n →
      if a[i+1] ≠ space → i, j, w[j+1] := i+1, j+1, a[i+1]
      [] a[i+1] = space → n := i
      fi
   od;
   writeword(w,j);
   n := n0;

   do i < n →
      if a[i+1] = space → i := i+1
      [] a[i+1] ≠ space → n := i
      fi
   od
$od;
$r: i = n0 => true
|p
```

According to our previously stated convention, because the loop has a $i: state its execution will begin in this state. This will avoid the need to write the text of the fi-mechanism twice, once at initialization and a second time in the body of the loop.

It is interesting to examine the details of the solution we have presented both with respect to "input" and output (input here refers to array accessing rather than a file read). With respect to input, initially, and with each successive iteration, a new word is "read" together with any trailing spaces, e.g.

Input: |THIS |TEXT |SEEMS |SHORT | (R1, R2, R3, R4)

In contrast with respect to output we have

Output: |THIS| TEXT| SEEMS| SHORT| (W1, W2, W3, W4)

It is also useful to make a comparison with the input and output sequences for Dijkstra's algorithm [14] given below.

I/O sequences for Dijkstra's Algorithm

Input: |THIS TEXT SEEMS SHORT | (R1, R2, R3, R4, R5)

Output: |THIS |TEXT |SEEMS |SHORT| (W1, W2, W3, W4)

With respect to input for Dijkstra's algorithm the fact that the first character of the next word must be read with each iteration is due to the input primitives chosen. We would suggest that as a principle of programming style it is better to "peak-ahead" as this allows us to maintain a more meaningful invariant. In our algorithm a single word with trailing spaces is "read" with each iteration whereas with Dijkstra's implementation the

second and subsequent characters of the current word are read together with any trailing spaces and the first character of the next word if there is one. Note that the situation is analogous to that for the unordered linear search discussed earlier.

For the output of our algorithm with *each* iteration a new word preceded by space is written. With Dijkstra's algorithm for each iteration a new word is written out together with a trailing space *if it is established that there is still another word to be processed.* We would suggest that the output invariant maintained by the former implementation is to be preferred because the structure of the algorithm is such that a space is written *unconditionally* with each iteration of the outermost loop. It might be added that Dijkstra has employed the same loop control structure as that traditionally employed for the partitioning problem discussed earlier with the consequence that extra testing must be employed in the body of the loop.

As it stands, our algorithm for the text formatting problem has two built-in inefficiencies that may be unacceptable to some when implemented in a deterministic language like Pascal. Fortunately these two inefficiencies can be easily eliminated without changing the basic structure of the algorithm.

The first inefficiency occurs in the first internal loop that has the job of accessing the characters in a word until a terminator is encountered. Whenever this loop terminates with "a[i+1] = space" true, it follows that i can be incremented by one before applying the second internal loop whose job it is to read trailing spaces.

The second inefficiency occurs with the second loop whenever it terminates with a[i+1] ≠ space true. This implies that the first character of the next word has been encountered. This can and should be accommodated when the *next* iteration is made. This latter refinement can most easily be incorporated by first applying a mechanism that strips off any leading spaces preceding the first word. Such an action sets up conditions that best allow us to apply the second refinement. These steps are left as an exercise to the reader.

Example 5.6 Text Scanning

Knuth in his structured programming discussion [ref. 15 p 271] cites an interesting text processing problem. Basically the problem requires the processing of a stream of text where we want to read and print the next character from the input. If a single '/' (slash) is encountered we need to advance to the next tab-stop position. However, if two consecutive slashes "//" are encountered the output should be advanced to the beginning of the next line. After printing a period "." an additional space should be inserted in the output.

The Π-mechanism in this case will simply be one that reads and writes one character at a time provided no slash has been encountered. After including the refinement that a "space" must be written after a period we may propose the following Π-mechanism:

```
i := eof, n := n0;

do i ≠ n ⟶
   read(x); i := eof;
   if x ≠ '/' ⟶ write(x); if x = '.' -> write(' ') fi
   [] x = '/' ⟶ n := i
   fi
od
```

What remains is to generalize this mechanism so that it acts appropriately when the Π-mechanism experiences forced termination. It is apparent that a Δ-mechanism is required. What the Δ-mechanism must accomplish, having encountered one '/' is to distinguish whether a tab or a newline should follow in the output. The Δ-mechanism

may therefore take the form:

```
read(x);
if x≠ '/' —> write(tab); write(x); if x = '.' -> write(' ') fi
[] x = '/' —> writeln
fi
```

We see that the basic structure of this algorithm follows the same underlying pattern as the previous example. We may therefore write the final implementation in the following way.

```
|p true —>
$vri.n.n0:boolean; x:char —>
   n0 := true;

$do i ≠ n@ —>
   read(x);
   if x ≠ '/' —> write(tab); write(x); if x = '.' -> write(' ') fi
   [] x = '/' —> writeln
   fi;

$i: i := eof, n := n@;

   do i ≠ n —>
      read(x); i := eof;
      if x ≠ '/' —> write(x); if x = '.' -> write(' ') fi
      [] x = '/' —> n := i
      fi
   od
$od;
$r: i = n@ ⟹ true
p|
```

In this implementation we have borrowed Pascal's I/O facilities.

It is difficult to make an absolute comparison of this mechanism with Knuth's implementation as at face value he does not take into account the possibility of encountering an end-of-file. The present construction avoids the problems encountered by Knuth when slashes (e.g. one or two consecutive slashes) are met in the input stream. The iterative ñ-mechanism, by simply reading and writing characters until either the input is exhausted or a slash is encountered, resolves the problem with the slashes in a more straightforward manner. We would suggest that present mechanism is cleaner and conceptually simpler than Knuth's implementation. Knuth argues against the idea of duplicating the code associated with the period. This can easily be avoided in our implementation too by making use of Pascal's buffered input facilities. There is, however, no significant gain to be made by pursuing such an alternative. We do not accept Knuth's argument that duplication of code is a "waste of energy". Methodologically we suggest that the Δ-mechanism's role is also to make progress towards termination. Hence the underlying structure of our mechanism.

Example 5.7 Exponentiation

The problem of computing $z = a^b$ where $a > 0$ and the integer $b \geq 0$ is a well-known problem. We will discuss it briefly because it raises several fundamental issues about the structure of loops and programming style. Choosing first a well-formed postcondition (which is not simply $z = a^b$) and then projecting the invariant we get

$$P: 0 \leq y \wedge z * x^y = a^b$$

In attempting to solve this problem we will need to change *at least* two variables in order to maintain the invariant. The most of obvious choice is to change z and y while maintaining the invariant. This leads directly to the simple method for exponentiation that is a linear function of exponent.

The other alternative, which turns out to be more interesting, is to instead change the pair of variables x, and y while maintaining the invariant. Under these conditions, admitting the operations of multiplication and division, the weakest iterative mechanism that can establish the postcondition will be one that deals with the case where b is a power of two. With the projection values x = a, y = b and z = 1 the ii-mechanism can *therefore take the form:*

```
do even(y) —>
   y := y div 2, x := x * x
od
```

This mechanism terminates with y odd and greater than or equal to one. If y has been reduced to one there is no need for further iteration and so the iterative mechanism can terminate. In the other case, where y is odd and greater than one it is necessary to derive the associated /\-mechanism that will shift the computation to a state where it is again possible to apply the ii-mechanism. Taking these steps through in a similar manner to that applied for previous examples we are led to the following implementation.

```
Ip 0 < a ^ 0 ≤ b —>
$vrx,y,z:integer —>
   x := a, y := b, z := 1;

$do 1 < y —>
   y := y-1, z := z * x;

$i: do even(y) —>
       y := y div 2, x := x * x

   od
$od;
$r: y = 1 => z := z * x, y = 0 => z
pI
```

With this implementation, once y has been reduced to 1, there is no need for furtner iteration. At this point we might be satisfied with the development given as it fits naturally into the same framework as the previous two algorithms which possess deductive dynamic initialization states. It is, however, useful to make comparisons with two other implementations, one given by Dijkstra [3 , p.66], and a second given by Gries [4 , p.240].

Dijkstra's implementation has the form.

```
x,y,z := a,b,1,
do y ≠ 0 —>
   do even(y) —>
      x, y := x * x, y div 2
   od,
   y, z := y - 1, z * x
od
```

This implementation is essentially the same as ours except that the order of the ĩi-mechanism and the other mechanism in the body of the loop has been reversed. Dijkstra suggests that the example above leads him to the conclusion that "support is weak" for the need to have loops with "intermediate exits".

We would argue that our implementation is to be preferred because it recognizes that the ĩi-mechanism *alone* can take the computation through to a state where it is possible to establish the postcondition (i.e. by z := z * x) without further contribution from the body of the loop. In contrast Dijkstra's implementation is always committed to apply *the complete Δ-mechanism once the loop is entered.* A further difference is that in our algorithm y is maintained as either even or odd whereas in Dijkstra's implementation it is reduced to zero which is not considered in mathematics at least to be an even number. Hence the loop does not maintain as an invariant property that y remains either even or odd. These distinctions may be too subtle for some but taken in conjunction with the other examples that require deductive initialization, it is apparent that there is a consistent pattern operating. The case where y = 1 requires a test for evenness with Dijkstra's implementation which is naturally avoided by our implementation.

A comparison with Gries' implementation given below raises another issue:

```
x,y,z := x,y,1;
do ⓪ < y ∧ even(y) ⟶ y,x := y div 2, x * x
[] ⓪ < y ∧ odd(y) ⟶ y,z := y-1, z * x
od
```

Because the tests for even and odd are incorporated into the loop guard the test ⓪ < y must be applied as many times as the even and odd tests. Consequently, the test ⓪ < y will, on average, be applied considerably more times than is necessary. Our implementation, with its ĩi-mechanism, avoids this problem.

There still remains an inefficiency in our implementation that has been overlooked. The inefficiency arises from the way in which the ii-mechanism and the Δ-mechanism have been coupled together. Part of the role of the Δ-mechanism is to change y from an odd value to an even value so that the iterative ii-mechanism can be re-applied. It follows that the test to see whether or not y is even is not needed until *after* the body of the ĩi-mechanism loop has been executed at least once. As such, our existing loop structure will, in general, make more applications of the test "even(y)" than are needed to solve the problem. The following program structure eliminates the redundant "even(y)" testing.

```
do even(y) ⟶
   y := y div 2, x := x * x
od;
$do 1 < y ⟶
   y := y-1, z := z * x;

   rp
      y := y div 2, x := x * x
   pr ⟶ odd(y)
$od;
$r: y = 1 ⇒ z := z * x, y = ⓪ ⟹ z
pl
```

In this implementation, we describe the Δ-mechanism and the ĩi-mechanism in the body of the loop as being *co-operatively coupled.* The price we have had to pay for this rationalization of the mechanism has been to explicitly include the guarded ĩi-mechanism as the initializing step and replace the guarded ĩi-mechanism in the loop body by its unguarded form (i.e. a repeat-loop). The general framework for the co-

operative coupling refinement is:

guarded ĩi-mechanism; {initialization}

$do guard →
  Δ-mechanism;

  unguarded ĩi-mechanism
$od

This refinement can be applied equally well to the previous two algorithms that involved deductive initialization *provided* the main loop guard is chosen such that the Δ-mechanism *cannot* establish the postcondition without subsequent application of the subsequent application of the unguarded ĩi-mechanism.

It should be noted that it is not possible to apply the co-operative coupling refinement to *implementations where the ii-mechanism precedes the Δ-mechanism in the* body of the loop. Furthermore, if we were to choose the guard of the main loop such that the Δ-mechanism in the loop body were able to establish the postcondition then it would not be possible to employ an unguarded ii-mechanism in the loop body. In our example this situation would arise if we had used the guard "$\phi < y$" instead of "$1 < y$".

We will in the next section see other problems with a higher degree of symmetry that can be treated according to essentially the same principles.

5.4. Iterative Initialization

There is an important class of loop problem that follows on naturally from the discussion in the preceding section. These are a set of problems, which, on the surface at least, appear to possess a measure of symmetry. They are often characterized by possessing more than one variable or collection of data to which it would appear that essentially the same mechanism could be applied in sequence. Probably the best way to introduce the issues raised by this class of problems is by way of example. We will now pursue this course.

Example 5.8 The Greatest Common Divisor Problem

The greatest common divisor problem should perhaps by now have been relegated to a position beyond discussion. We will attempt to present a different perspective on the development, interpretation, and implementation of solutions to the problem in the light of the proposed methodology.

We may assume that we are given two positive integers, a and b, and we are required to establish the postcondition:

$$R: x = y = \gcd(x,y) \wedge \gcd(x,y) = \gcd(a,b) \wedge x > \phi \wedge y > \phi$$

The ranges for the variables x and y are

$$x: 1 \leq x \leq a$$
$$y: 1 \leq y \leq b$$

as a, and b must at least share the divisor 1. We are free to choose as our projection values x = a and y = b from which we are led to the following invariant and determinate:

$$P: \gcd(x,y) = \gcd(a,b) \wedge x > \phi \wedge y > \phi$$

$$D: x = y = \gcd(x,y)$$

In formulating our solution to this problem in the first instance we are looking for the weakest general iterative mechanism capable of establishing the postcondition. There

are two variables associated with the problem, however, in the first instance we are looking for the weakest mechanism so we need to investigate whether there exists a mechanism that possesses the following characteristics - it changes only one variable while maintaining the invariant and it is iterative and capable of establishing the postcondition. These requirements correspond directly to the situation where one of the variables is a *multiple* of the other. The corresponding $\bar{\Pi}$-mechanism might therefore take the form

```
do x > y ⟶ x := x - y od
```

This mechanism can terminate either with x = y, confirming that y is a multiple of x, or with x less than y. If the latter case prevails further processing will be required. At this point in the development the skeleton structure that we have for the gcd algorithm is:

```
rp
   Δ-mechanism: ???

   do x > y ⟶ x := x - y od
pr ⟶ x = y;
```

The nature of the dynamic initialization in the loop body for the $\bar{\Pi}$-mechanism may not be immediately apparent. However, its role is by definition to change the state of the computation to a point where it is again possible to attempt to apply the $\bar{\Pi}$-mechanism again. Examining the conditions that apply after termination of the $\bar{\Pi}$-mechanism, it follows that x must be *less* than y if further processing is required - this is the opposite situation to that which the $\bar{\Pi}$-mechanism was designed to handle. As x has been *changed as much as is possible we might expect that the Δ-mechanism will need to* change y. One possibility would be to use

```
y := y - x
```

This mechanism however *cannot* guarantee to change the state of the computation in *one* iteration to a situation where $x \geq y$ and it is again appropriate to apply the $\bar{\Pi}$-mechanism. The obvious choice of Δ-mechanism is therefore one which has a greater iterative strength i.e.

```
rp
   y := y - x
pr ⟶ x ≥ y
```

Our complete implementation of the gcd algorithm can therefore take the form:

```
ip ϕ < a ∧ ϕ < b ⟶
$vrx,y:integer ⟶
   x := a, y := b;

$rp
   rp
      y := y - x
   pr ⟶ x ≥ y;

$i: do x > y ⟶
      x := x - y
   od
$pr ⟶ x = y
$r: x = y ⟹ x
pi
```

With this implementation we have deliberately chosen to use a repeat loop for the Δ-mechanism, because, given that the mechanism always starts in the $i: state, to achieve co-operative coupling between the two loops an unguarded Δ-mechanism is required. This repeat loop/do loop structure can be used widely for algorithms with an iterative Δ-mechanism.

Several comments are in order about this implementation of the gcd algorithm. As a reference point we should note a familiar implementation of the gcd algorithm which has the form

```
x := a, y := b;
do x ≠ y ⟶
   if x > y ⟶ x := x - y
   [] y > x ⟶ y := y - x
   fi
od
```

The advantage of our algorithm over this implementation lies in the fact that the iterative ii and Δ-mechanisms ensure that on average the outer loop guard for our algorithm will be executed significantly fewer times than the guard "x ≠ y". We would therefore suggest that the implementation with two loops in the loop body is the preferred solution. Another familiar implementation that possesses two iterative mechanisms in the loop body has the form:

```
x := a, y := b;
do x ≠ y ⟶
   do x > y ⟶ x := x - y od;
   do y > x ⟶ y := y - x od
od
```

In this implementation the two inner loops are not co-operatively coupled with consequence that the test "x > y" is applied more times than are necessary to solve the problem i.e. *after* the first iteration of the outer loop x will always be greater than y if the outer loop is still active. This inefficiency cannot be overcome by changing the first inner loop (or for that matter, the second) to a repeat loop. In fact, it would seem that the only way to achieve co-operative coupling and hence eliminate the inefficiency for this mechanism is to admit a $i: state in which the algorithm begins execution.

What is probably most interesting about this algorithm is the way in which the proposed methodology led us directly to the preferred solution. Furthermore, even without a formal specification of the postcondition in searching for the weakest iterative mechanism we would be led down the same path. What this example underlines is the constructive power of the methodology. We will now turn to another example which also by now should have been relegated to a position beyond discussion.

Example 5.9 Two-way Merge

The two-way merge problem is more usually concerned with files. We will, however, phrase the problem in terms of arrays. The file version, if needed, can easily be obtained using the methods suggested in section 4. Our concern is to merge two ordered arrays a[1..mϕ] and b[1..nϕ] to produce a third array c[1..mϕ + nϕ]. Our solution should handle all cases including those where one or both of the input arrays are empty.

At this point we could write down formal postconditions, derive invariants and proceed much in the same way as for most of our other examples. We have, however, chosen to proceed more informally, partly to illustrate that the methodology does not necessarily rely heavily on the more formal apparatus.

At any point in time we will assume that a[1..i] and b[1..j] are merged to give c[1..k], where k = i+j. In carrying out the merge the possible variables for change are, i, j, k, and c[k+1], the first three of which would be initially zero. It is not possible to carry out a merge by changing either one or two of these variables. Our weakest iterative mechanism will therefore be one that changes at least three of these variables. The two choices are either i, k, and c[k+1] or j, k and c[k+1]. Selecting the first choice, the weakest iterative mechanism that could reduce the problem to one where no further merging would be required is of the form:

```
i := 0, j := 0, m := m0;
do i < m →
   if a[i+1] ≤ b[j+1] → i, k, c[k+1] := i+1, k+1, a[i+1]
   [] a[i+1] > b[j+1] → m := i
   fi
od
```

This mechanism corresponds to that situation where all the a[1..m0] occur in the merged output *before* the first element in the b array. The mechanism is designed to terminate as soon as it detects other than the assumed most favourable situation in accordance with the standard practice we have adopted for constructing ii-mechanisms.

We need to understand clearly what the implications are of the ii-mechanism terminating in either of the two possible states. Whenever the ii-mechanism terminates *naturally* with i = m = m0 all the a[1..m0] will have been merged. Consequently, the problem that remains is *not* a merging problem but rather one of *copying* the remaining elements b[j+1..n0] *into the output array C. The implication* of this for our implementation is that the loop controlling the complete merging process should terminate as soon as it is detected that there is no longer a defined merging problem remaining. *Forced termination* of the ii-mechanism in its other state (ie with i < m0) suggests that some form of delta mechanism is required. Our immediate suggestion for the Δ-mechanism might be:

```
j, k, c[k+1] := j+1, k+1, b[j+1]
```

However, as this cannot guarantee to place the computation in a state where it may be most appropriate to apply the ii-mechanism a better suggestion in accordance with cooperative coupling is:

```
rp
   if b[j+1] ≤ a[i+1] → j, k, c[k+1] := j+1, k+1, b[j+1]
   [] b[j+1] > a[i+1] → n := j
   fi
pr → j = n
```

Our basic structure for the merging component of the problem *might* therefore take the form:

```
do ? →
   rp
      if b[j+1] ≤ a[i+1] → j,k,c[k+1] := j+1, k+1, b[j+1]
      [] b[j+1] > a[i+1] → n := j
      fi
   pr → j = n;

   do i < m →
      if a[i+1] ≤ b[j+1] → i,k,c[k+1] := i+1, k+1, a[i+1]
      [] a[i+1] > b[j+1] → m := i
      fi
   od
od
```

The problem that remains with the merging part of the implementation is to control the termination of the mechanism when the problem ceases to be defined as a merging problem. It is clear that either of the internal loops has the potential to terminate *naturally* and hence bring about a state of the computation where the problem has been reduced to a copying problem. For example, the repeat-loop can terminate with $j = n = n_0$ and $i < m$ which implies that a[i+1..m] remains to be copied. In this situation the guard on the do-loop that follows it (i.e. $i < m$) is *not sufficient* to prevent the subsequent application of the test

```
if a[i+1] ≤ b[j+1]...
```

which will be out of bounds with respect to the b array. To overcome this the repeat-loop mechanism must be able to communicate to the do-loop (and vice versa) that a merging problem is no longer defined.

An effective way to accomplish this communication is as follows:

```
m := m0, n := j;
$rp
   rp
      if b[j+1] ≤ a[i+1] → j,k,c[k+1] := j+1, k+1, b[j+1]
      [] b[j+1] > a[i+1] → n := j, m := m0
      fi
   pr → j = n;

$i: do i < m →
      if a[i+1] ≤ b[j+1] → i,k,c[k+1] := i+1, k+1, a[i+1]
      [] a[i+1] > b[j+1] → m := i, n := n0
      fi
   od
$pr → j = n
```

The way this communication mechanism works is as follows. Each merging loop uses a *switch* to control whether its successor can continue to execute. Setting $n = j$ initially effectively "turns off" the repeat-loop. If the inner do-loop terminates naturally then the merging mechanism terminates. If, however, the do-loop is subject to *forced* termination (i.e. by the assignment $m := i$, indicating that an element from the b-array should be merged next) then the repeat-loop mechanism is "switched on" by the assignment $n := n_0$. The do-loop mechanism can only be subsequently switched on by the repeat-loop if it has a forced termination indicating the requirement for further merging. And so the two iterative components that carry out the merging are alternatively switched on and off in each case, depending on how their predecessor terminates. This control structure is a general one that can be used much more widely

than the present application. A repeat-loop has been deliberately used for the outermost loop because it will allow the compiled implementation to execute fewer steps as the mechanism will "fall through" when the guard is true.

We are now left with the task of implementing the copying mechanism that must follow the termination of the actual merge part of the process. This aspect of the implementation is straight forward and so we are led directly to the final implementation.

```
lp 1 ≤ mϕ V a ≤ nϕ ⟶
$vr i,j,k:integer; c[1..mϕ + nϕ]:integer ⟶
   i := ϕ, j := ϕ, k := ϕ;

   lp 1 ≤ mϕ ^ 1 ≤ nϕ ⟶
   $vr m,n:integer ⟶
      m := mϕ, n := j;
   $rp
      .rp
         if b[j+1] ≤ a[i+1] ⟶ j,k,c[k+1] := j+1,k+1,b[j+1]
         [] b[j+1] > a[i+1] ⟶ n := j, m := mϕ
         fi
      pr ⟶ j = n

   $i: do i < m ⟶
         if a[i+1] ≤ b[j+1] ⟶ i,k,c[k+1] := i+1,k+1,a[i+1]
         [] a[i+1] > b[j+1] ⟶ m := i, n := nϕ
         fi
      od
   $pr ⟶ j = n
   $r:  (i = mϕ ^ j < nϕ) V (i < mϕ ^ j = nϕ) ⟹ i,j,k,c
   pl;

   do i < mϕ ⟶ i,k,c[k+1] := i+1, k+1, a[k+1] od;
   do j < nϕ ⟶ j,k,c[k+1] := j+1, k+1, b[j+1] od;
$r: i = mϕ ^ j = nϕ ⟹ k,c
pl
```

When we compare this implementation of the two-way merge with other alternatives it is immediately apparent that it is textually more complex. We may, therefore, ask what is the justification for this increased textual complexity? The justification in part comes from the fact that the implementation is mechanistically simpler than other alternatives. Consider, for example, the commonly used merging construct:

```
do i < m ^ j < n ⟶
   if a[i+1] ≤ b[j+1] ⟶ i,k,c[k+1] := i+1, k+1, a[i+1]
   [] a[i+1] > b[j+1] ⟶ j,k,c[k+1] := j+1, k+1, b[j+1]
   fi
od;
"copying operations"
```

For this implementation, with each iteration tests on both i < m and j < m must be made. In our implementation only one of these tests need to be made. The "overhead" in our implementation caused by the need to employ forced termination and define the merging operation dynamically should in general be offset by the capacity to use simpler mechanisms to merge sequences of a values before the current b value and vice versa.

Dijkstra [ref. 3 p. 126] suggests another merging mechanism that is textually simpler than either of the above proposals. The basic control structure for this implementation using some set notation is:

```
x,y,z := X,Y,∅;
do x ≠ ∅ or y ≠ ∅ →
   "transfer an element from (x+y) to z"
od
```

where x+y refers to a union of two sets and ∅ identifies the empty set. Although an attractive implementation, mechanistically it relies on the use of sentinels (which is not always practical) and furthermore it does not degenerate to a simpler mechanism when the problem reverts to a copying problem. The natural structure of the problem is therefore hidden by the mechanism.

Of course, it is easy enough to question our arguments about the merging problem. What is, however, much more important is to recognize the underlying characteristics of solutions derived by always seeking and building solutions to problems around the weakest iterative mechanism that can establish the postcondition. We will come back to these issues in the section on classification of loop constructs.

One final comment is in order before we leave our discussion of the merging algorithm. Our implementation could in some contexts be criticized for not taking more effective action when either of the internal loops experiences forced termination. For example, with the do-loop when a[i+1] > b[j+1] we proceed as follows:

```
[] a[i+1] > b[j+1] → m := i, n := n∅
```

However, this condition invites us to apply a merge of the element b[j+1] i.e.

```
[] a[i+1] > b[j+1] → m,n,j,k,c[k+1] := i,n∅,j+1,k+1,b[j+1]
```

A similar strategy could be applied with the other merging loop with the attendant structural implication.

Example 5.10 Binary Tree Searching

To complete this section we will consider two more examples that raise certain issues about the application of the methodology we have introduced.

In considering the binary tree search for comparison we will focus on the representation employed by Knuth [15]. The binary search tree is represented by three arrays, a[i] denotes the information stored at node number i, and l[i] and r[i] are the respective node numbers for the roots of that node's left and right subtrees with empty subtrees being represented by zero. If we pursue the idea of looking for the weakest iterative mechanism we are quickly led to the following implementation with cooperative coupling and a deductive iterative initialization scheme.

```
ip ⊕ ≤ i ⟶
$vr p,n,n⊕:integer ⟶
   p := ⊕, n⊕ := ⊕;

$rp
   n := n⊕;
   rp
      if a[i] > x ⟶ p := i; i := l[i]
      [] a[i] ≤ x ⟶ n := i
   pr ⟶ i = n;

$i: n := n⊕;
   do i ≠ n ⟶
      if a[i] ≤ x ⟶ p := i; i := r[i]
      [] a[i] > x ⟶ n := i
      fi
   od
$pr ⟶ i = n⊕;
$r: i = n⊕ ⟹ p
pi
```

Before evaluation our first solution we shall present a textually simpler solution, the essential part of which has the form:

```
n := ⊕; p := n;

do i ≠ n ⟶
   if a[i] ≤ x ⟶ p := i; i := r[i]
   [] a[i] > x ⟶ p := i; i := l[i]
   fi
od
```

Comparing the two solutions we see that the $\overline{\Pi}$-mechanism has essentially the same loop and guard structure as our second solution. If we had in fact applied the same type of refinement as finally suggested for the two way merge we would have ended up with the following ii-mechanism

```
n := n⊕;
do i ≠ n ⟶
   if a[i] ≤ x ⟶ p := i; i := r[i]
   [] a[i] > x ⟶ n := i; p := i; i := l[i]
   fi
od
```

which is just our second solution with the statement n := i inserted to bring about forced termination.

Study of the two solutions suggests that the second solution is to be preferred because it avoids the overhead needed to bring about forced termination of both the $\overline{\Pi}$-mechanism and the Δ-mechanism. We must now ask what are the implications of this most recent result. It would seem to suggest that the strategy of always looking for the weakest iterative mechanism that can establish R does not always lead us to the preferred implementation. However, all is not lost. The situation we have just encountered is signalled by the circumstance where the guard (or its complement) for the outermost loop is identical to the guard for the iterative $\overline{\Pi}$-mechanism. In such instances we should evaluate the situation to decide whether or not it is appropriate to

carry out the mechanical transformation needed to arrive at the second preferred solution to the problem.

At this point in the discussion it is convenient to introduce a little terminology that will make it easier to usefully evaluate the phenomena we have just encountered. We say that *iterative slack* exists in an implementation when it is possible to introduce one or more iterative constructs into the mechanism that will on average reduce the overall loop-guard-testing required by the implementation. This concept makes way for transformations that may allow on the one hand a speed-up of existing implementations while at the same time helping us to design *iteratively tight* implementations in the first instance. An implementation is said to be *iteratively tight* if it is not possible to introduce an iterative mechanism that will reduce the overall guard testing required by the algorithm. What we suggest is that the Π-mechanism methodology will in general provide us with mechanisms in which there is either no iterative slack or in which it has been minimized. In applying the methodology we must be aware of conditions that may suggest the preference for a non-iterative Π-mechanism.

The reader is left to draw his/her own conclusions about comparison with Knuth's two implementations. In our implementations we have chosen to separate the tasks of insertion and search. We have also taken precautions to handle the insertion of the first element in the tree. As a fundamental principle of loop construction we would suggest that any step(s) that are only applied at the last iteration should be separated from the body of the loop. We referred to this principle earlier as the *law of separation of concerns.*

Before leaving the tree search problem it is worth noting that the weakest iterative mechanism did not completely mislead us in pointing towards the "best" solution if we are prepared to resort to *sentinels (as Wirth does in ref. [16] p 201). What Wirth suggests* is essentially

```
s ↑.key := x; (sentinel)
while t↑.key ≠ x do
    if x < t↑.key ⟶ t := t↑.left
    [] x ≥ t↑.key ⟶ t := t↑.right
    fi
```

This mechanism requires all leaf nodes to be linked to a common node where the sentinel gets placed before the commencement of the search.

If one is to resort to these tactics we would suggest the following alternative implementation which is based on the application of the weakest iterative mechanism that could establish the postcondition *(we have kept with Knuth's representation).*

```
a[s] := x; (sentinel)

$do   a[i] ≠ x ⟶

    rp p := i; i := r[i] pr ⟶ a[i] ≥ x;
$i: do a[i] > x ⟶ p := i; i := p[i] od

$od
```

This implementation will on average make *less* applications of the test "a[i] ≠ x" than Wirth's implementation.

Example 5.11 Maximum Finding

We will now make one final diversion before moving on to issues related to the classification of loop mechanisms.

The problem we wish to consider is that of finding the maximum in an array of nϕ integers. This problem once again addresses the issues raised by the tree searching example. We will develop the solution informally as even the more formal development is very straightforward. There are two variables associated with the problem which can be changed; they are the array index i, and the maximum variable max. The weakest iterative mechanism which can establish the postcondition will be one that merely *confirms* (by changing "i") that a given candidate for the maximum is indeed the maximum in the array. Our ii-mechanism can therefore take the form

```
max,i,n := a[1],1,n$\phi$;

do i < n ⟶
   if a[i+1] ≤ max ⟶ i := i+1
   [] a[i+1] > max ⟶ n := i
   fi
od
```

Clearly the ii-mechanism as it stands cannot solve the general problem. Accommodating this fact we are led to the final implementation that selects, and then tries to confirm, successively better candidates for the maximum in the set.

```
ip $\phi$ < n$\phi$ ⟶
$vri,max,n:integer ⟶
   i := $\phi$

$do i < n$\phi$ ⟶
   max,i,n := a[i+1],i+1,n$\phi$;

   do i < n ⟶
      if a[i+1] ≤ max ⟶ i := i+1
      [] a[i+1] > max ⟶ n := i
      fi
   od
$od;
$r: i = n$\phi$ ⟹ max
pi
```

Like in our previous example, we note that the guards for the outer loop and the ii-mechanism are essentially the same. Furthermore we see that the ii-mechanism itself embodies a solution to the general problem is we include the step that takes full advantage of the condition *a[i+1] > max*. With these observations we are led to the alternative solution whose central mechanism has the form:

```
i, max := 1, a[1];

do i < n ⟶
   if a[i+1] ≤ max ⟶ i := i+1
   [] a[i+1] > max ⟶ i,max := i+1, a[i+1]
   fi
od
```

As for our previous example, we see that this mechanism is iteratively tight and so is to be preferred over our original solution.

Once again, if we choose to resort to the use of sentinels we can arrive at the implementation given below which can exploit the ñ-mechanism.

```
i := 0;

do i < n →
   i,max,a[n+1] := i+1, a[i+1], a[i+1];

   do a[i+1] < max → i := i+1 od
od
```

Knuth has shown [17] that the maximum only gets updated on average O(log n) times, consequently with this implementation the guard "i < n" is only tested on average O(log n) rather than n times as in our two previous algorithms. Doing a statement count with the Berkeley Pascal compiler for n = 1000 we find that for random data on average 3010 statements are executed for the second implementation while the third implementation only executes 2023 statements.

From the last two examples we have considered it should not be inferred that we are advocating the use of sentinels. What did come as a surprise from this investigation was that application of the ñ-mechanism methodology in a straightforward way led to a solution to the problem of finding the maximum that the author had not previously contemplated. Others, unaware of the methodology, have also been at a loss to discover a more efficient implementation than the conventional one.

6. On the Characterization of Loop Mechanisms

What is apparent from our study of loops is that there appears to exist a set of basic control mechanisms that can be widely used to solve problems that require iterative solution.

Rather than leave our discussion of loop mechanisms in its present state, it would seem appropriate to examine the prospect of formalizing this knowledge in some systematic manner.

What our investigation has done has been to indicate that there is possibly a deeper structure associated with loops that we have hitherto not seriously exploited.

In attempting to come to terms with this hypothesized deeper structure it would seem prudent to be aware of Chomsky's seminel work in linguistics [18,19]. What Chomsky has been able to do is provide a framework for the construction of theories of language. Within this framework his most important contribution has involved the specification of rules underlying the construction of sentences [18] using phrase structure and transformational grammars. The relevance of Chomsky's work in particular and formal language theory in general, has long been recognized in computing in the theory of compiler construction [20]. We do not, however, find the application of principles at least similar in intent to Chomsky's in programming methodology. If it is possible to accomplish such a task then one might hope for a somewhat similar set of benefits to those derived from having a grammatical theory for the construction of natural language.

The simplest and perhaps most obvious benefit from specifically identifying a higher level syntactic framework would be a taxonometric one. This would assist the systematic study of the discipline. We might also expect that an awareness of such a higher level framework should aid the semantic analysis of programs. Explicit knowledge of these rules should also improve a programmer's "competence" (in Chomsky's sense) since initially, at least, (because programming does not involve a natural language), he or she does not have any "internalized" rules for program construction of the type that Chomsky claims that natural language speakers possess.

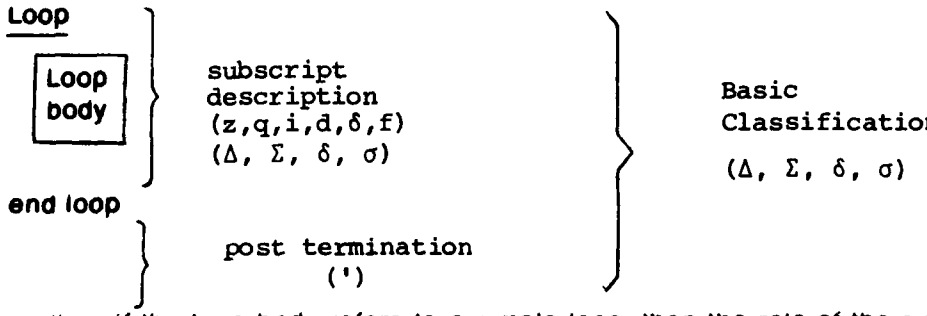
To attempt to do justice to a grammatical theory of program construction is beyond the scope and intent of the present discussion. Only a brief outline and exploratory investigation of the possibilities that we have raised will therefore be made at this point.

In attempting to draw on parallels with results in another discipline there is always the danger of introducing structure and terminology that is of questionable relevance. We will start this discussion by considering the simplest problem, that of inventing a classification discipline for simple and complex loop structures.

6.1. Classification of Loop Structures

There are many schemes that could be selected to classify loops. We have chosen a very simple scheme that will place loop mechanisms into a relatively small number of broad categories (this scheme is derived from, but independent of, the more detailed phrase structure scheme to be described in the next section). There are two categories of loops, *simple* (denoted either by δ or σ) and *complex* (denoted by Δ or Σ). A simple loop mechanism consists of a single loop whereas a complex loop refers to a nested loop structure. A "δ" is used to indicate a loop mechanism; the *subjective* variables (i.e. the required "output") of the loop are conditionally (or deductively) derived by the loop mechanism (e.g. a loop for finding the maximum in an array). A "σ" is used for loops that derive their results inductively (e.g. a loop for summing the elements in an array). A "Δ" is used for a nested loop that conditionally derives its subjective variables where a "Σ" is used for a loop with similar properties to a "σ" loop (eg standard matrix multiplication).

Subscripts are used to identify the structure of the loop body and a "'" superscripted "'" indicates the presence of a post termination state. A simple classification scheme with attached description alphabets might take the following form.



By convention, if the loop body refers to a simple loop, then the role of the subscripts is to describe how the subjective element(s) in the loop body are derived (note variables used to force termination are not considered).

The conventions used are shown in the following table:

Descriptor	Derivation
z	static
i	inductive
d	deductive
f	functional

The descriptive power is increasing (i.e. f has highest preference).

For complex loops the loop body is considered to have an initialization component, a simple or complex loop, and a possible post termination mechanism. The initialization component which forms the left-most component of the subscript follows the initialization conventions identified in section 5 and employs the descriptors in the table above (allowing also for iterative initialization). The internal loop(s) is described

using the basic loop classification scheme with the convention that subscripts are dropped to maintain just a two level scheme.

These simple conventions are sufficient to classify most (but not all) simple loop mechanisms. Using these conventions the classifications for some of the examples discussed in this paper are:

Algorithm	*Classification*
Integer Square Root	σ_i
Quotient Remainder	σ_i
Binary Search	δ'_d
Linear Search	δ'_d
Searching a two-dim. array	$\Delta'_{z\delta}$
Inventory Report	$\Delta_{i\delta}$
Prime Factorization	$\Delta_{f\sigma'}$
Partitioning an Array	$\Delta_{d\delta}$
Text Formatting	$\Delta_{f\delta'}$
Greatest Common Divisor	$\Delta_{\delta\delta}$
Two way Merge	$\Delta'_{\delta\delta}$

It is apparent that the description provided by the conventions that we have adopted in this section can only loosely classify loop mechanisms. We must therefore turn to the invention of a device, whose role is somewhat akin to Chomsky's phrase structure grammar [18] in order to more accurately characterize the structure of loop mechanisms.

6.2. A Loop Grammar

The intent in proposing a loop grammar is to attempt to identify a higher level underlying syntactic structure that could be employed in the construction of loops. There are possibly many ways to invent such a grammar. We will employ a scheme that follows on fairly naturally from the conventions followed in the preceding section.

As a simple example of a possible loop grammar we may propose the following *rewrite rules.*

$L \longrightarrow P + S$

$P \longrightarrow (a) + (z) + (i) + (d) + (f)$

$S \longrightarrow \delta + (R)$

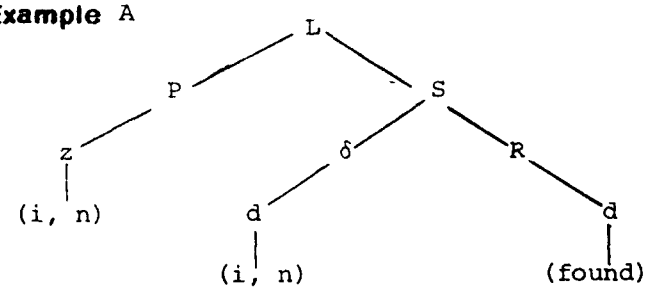
$\delta \longrightarrow (P)+(S)$

$R \longrightarrow (z) + (i) + (d) + (f)$

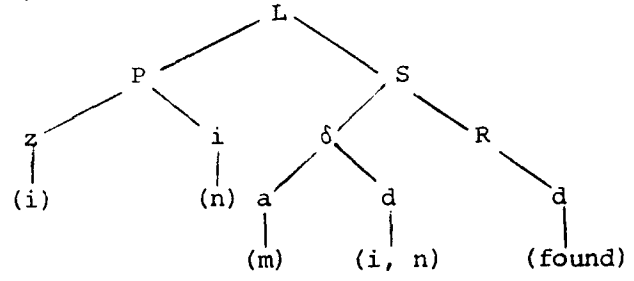
The initial loop structure is L. S is the accompanying loop body. P refers to the initialization segment and R to the post-termination segment. The "a" is used to deal with auxiliary variables. The other components identified in the rewrite rules have been previously defined. Constituent analysis of the loop structure is performed textually top-to-bottom.

The terminal strings are the variables identified in the loop mechanism. Some examples of applying this simple grammar are as follows (we have used examples previously discussed in the text)› It should be noted that the rewrite rules we have proposed constitute only a *weak* grammar in that they do not necessarily exclude incorrectly formulated loop mechanisms.

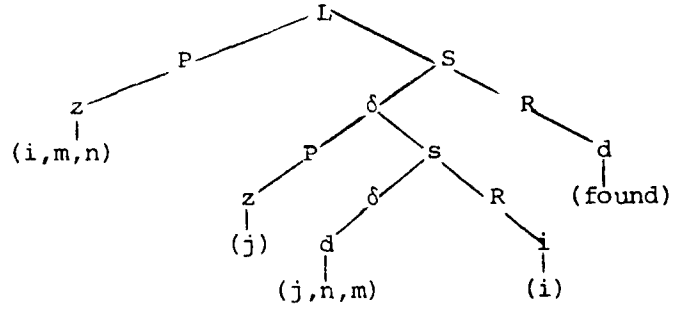
Linear Search - Example A



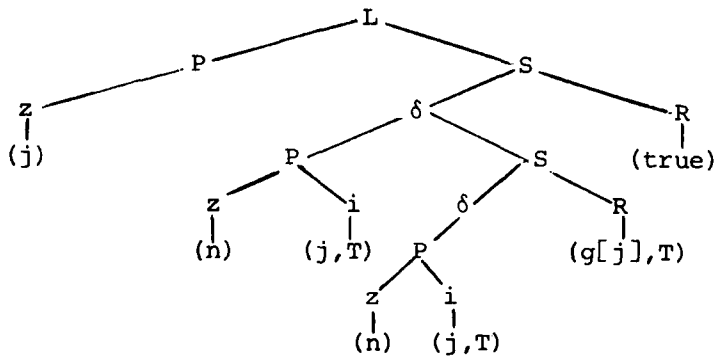
Binary Search - Example B



Two-dimensional Search - Example C



Inventory Report Example D



These examples should be sufficient to indicate how a loop grammar might be applied. Of course a much more extensive study would be needed to implement a loop grammar of sufficient power to be of real practical value.

At this point our analysis of loop structures is still not complete as we have yet to identify useful transformations.

6.3. Loop Transformations

What is apparent from any extensive and detailed study of loop mechanisms is that a small number of generic loop structures account for a large number of problems. This observation is not necessarily apparent from a casual survey of the literature but rather it is a suggestion of what could be achieved by application of principles like the law of separation of concerns and other aspects of a good discipline for loop construction.

Within the framework of basic loop structures we find that there are two mechanisms that dominate. The simpler mechanism is one that maintains its loop invariant either conditionally or unconditionally throughout its iterative phase by application of what is essentially a single mechanism within the loop body.

The other dominant mechanism is one in which there are essentially two components to the loop body, one component that maintains the loop invariant in a relatively straightforward way (much like the simpler mechanism) and a second often more complex component that needs to be applied when the simpler component is unable to maintain the loop invariant. This second component also has the role of maintaining the loop invariant but it must do so by application of a more complex (or at least different) mechanism. We will now explore several useful transformations that can be used to arrive at different implementations of these basic loop mechanisms.

Consider first the basic loop mechanism:

```
do B1 ⟶
   if C1 ⟶ S1
   ▯ C2 ⟶ S2          ..(6.1)
   fi
od
```

where the condition C2 is usually weaker than ¬C1. If it is appropriate to apply C1 directly as a guard we could increase the iterative strength of this component by turning it into a loop thereby removing the need to guard S2 within the body of the loop. The transformation yields

```
do B1 ⟶
   S2;
$i: do C1 ⟶ S1 od        ..(6.2)
od
```

This transformation takes up some of the iterative slack in the original mechanism. It can be appropriately applied to problems like Fermat's Factoring algorithm [12], and the prime factorization algorithm where the mechanism S1 needs to (or may need to) be applied on average considerably more times than S2 to complete the computation. We will illustrate the transformation for the gcd algorithm.

Original Algorithm

```
do a ≠ b ⟶
   if a > b ⟶ a := a - b
   [] a < b ⟶ := b - a
   fi
od
```

Transformed Implementation

```
do a ≠ b ⟶
   b := b - a

$i: do a > b ⟶ a := a - b od
od
```

We call this an *asymmetrical transformation.*

For problems like the gcd algorithm even the transformed implementation still possesses some iterative slack because the S2 mechanism may not be iteratively strong enough to bring the computation into a state where it is most appropriate to apply the internal loop. This can be overcome by increasing the iterative strength of *both* alternative constructs in the original algorithm - this amounts to a *symmetric transformation.* This second transformation is given below.

Symmetric Transformation

```
do B1 ⟶
   rp S1 pr ⟶ C1 V ¬B1;..(6.3)
$i: do C1 ⟶ S2 od
od
```

Applying this transformation to the original gcd algorithm we get

```
do a ≠ b ⟶
   rp b := b - a pr ⟶ b ≤ a;
$i: do a > b ⟶ a := a - b od
od
```

This implementation is iteratively tight. As we have seen in our discussion of a tree-searching algorithm it is *not always* appropriate to make such transformations - i.e. when the original mechanism is already iteratively tight. It is important to recognize that these transformations are *reversible.* Our constructive methodology will yield loop bodies with the greatest iterative strength (i.e. mechanisms that are *iteratively resolved).* If circumstances demand, such solutions can be transformed to an implementation more appropriate for the problem.

When the conditions C1 and C2 in (6.1) are not appropriate for use as guards then the transformations become slightly more complex than (6.2) and (6.3) although they may be thought to form analogs of (6.2) and (6.3) respectively.

The simplest of the cases, which is often encountered in file processing, has the following iteratively resolved representation:

```
P1;
do B1 ⟶
   P2; F2;

   do B2 ⟶
      if C2 ⟶ S2          ..(6.4)
      [] ¬C2 ⟶ F2'
      fi
   od;
   R2
od
```

Note that in this simplest representation P2 is an inductive initialization for the internal loop, the guards B1, and B2 are equivalent, and F2 and F2' are the elements of the mechanism needed to force termination of the internal loop. The inventory report problem is an example of a mechanism that possesses this basic underlying loop structure. The corresponding iteratively unresolved mechanism has the following form:

```
P1; P2;
do B1 ⟶
   if C2 ⟶ S2
   [] ¬C2 ⟶ R2; P2        ..(6.5)
   fi
od;
R2
```

Rewriting the inventory report implementation in this form we have:

```
j := 0;
j, T := j+1, t[j+1];

do j< n0 ⟶
   if g[j] = g[j+1] ⟶ j, T := j+1, T + t[j+1]
   [] g[j] ≠ g[j+1] ⟶ write (g[j], T); j, T := j+1, t[j+1]
   fi
od;
write(g[j], T)
```

The mechanism (6.5) reverts to a simpler form when there is no post-termination mechanism R2 for the internal loop. The iteratively resolved maximum finding algorithm has these underlying characteristics. Its corresponding iteratively unresolved mechanism has the form:

```
P1; P2
do B1 ⟶
   if C2 ⟶ S2
   [] ¬C2 ⟶ P2            ..(6.6)
   fi
od
```

The underlying reason for the difference between (6.5) and (6.6) is that in (6.6) the initialization P2 is capable of establishing the post-condition whereas in (6.5) the corresponding P2 is not capable alone of establishing the post-condition i.e. P2 *followed by* R2 is needed to establish the postcondition. Such distinctions as that between 6.5 and 6.6 indicate the value of loop transformations.

It should be pointed out that Bergland gives an implementation of the inventory report problem [11] which seemingly avoids the need for a post-termination mechanism. This happens only because he employs what is effectively a sentinel and so his solution is not really a general solution for the problem.

When the dynamic initialization for the internal loop mechanism has a deductive component the iteratively resolved form is:

```
P1;
do B1 →
   D2; S2; F2

   do VB2 →
      if C2 → S2
      [] ¬C2 → F2'          ..(6.7)
      fi
   od
od
```

and the unresolved form is:

```
P1;
do B1 →
   if C2 → S2
   [] ¬C2 → D2; S2          ..(6.8)
   fi
od
```

There may also be a D2 component after termination. If there is more than one internal loop in sequence then the transformations are slightly more complex.

We have now covered most of the asymmetric transformations. The symmetric analog of 6.3 is needed where it is not appropriate to use C1 as a guard.

The iteratively resolved symmetric transformation has the form:

```
P1; F2;
do B1 →
   rp
      if C2 → S2
      [] ¬C2 → F2
      fi
   pr → ¬B2;                ..(6.9)
$i: do B2' →
      if ¬C2 → S2'
      [] C2 → F2'
      fi
   od
od
R1
```

The unresolved symmetric form is:

```
P1;
do B2 ∧ B2' ⟶
   if C2 ⟶ S2
   [] ¬C2 ⟶ S2'          ..(6.10)
   fi
od;
R1
```

The two-way merge is a good example that exhibits both these forms.

We have now covered transformations for most of the simple cases that we have explored in this paper. The loop designer should be familiar with all these forms and have an understanding of when it is appropriate (and when it is not) to introduce forced termination and iteratively resolved mechanisms.

In concluding our discussion of loop transformations several things are apparent. Firstly, this subject deserves a deeper and more extensive treatment than has been given here as a fully developed theory of loop transformations would put programming methodology on a much sounder footing. Such a study would, hopefully, shed more light on the connection between the syntactic structure of loop mechanisms and their underlying semantics. Even the present treatment can tell us such things as that the post-termination mechanism of an internal loop holds the key to the purpose of the enclosing external loop. Also where there is a ⫫-mechanism and no post-termination state in the loop body then we can usually work out from the ⫫-mechanism *alone* what is the purpose of the loop mechanism - the ⫫-mechanism for the partitioning algorithm illustrates this point rather well. Pursuing such a course we are inevitably confronted with the question "should we search for *deep structures* of loop mechanisms along the lines of that pursued and disputed by linguists in their study of natural language?"

7. Conclusions

Jackson's data structure design, Dijkstra's methodology, or any other methodology, including the present one is only useful so long as one applies it not as a believer, but as a rational critic. Floyd, Hoare, Dijkstra, Wirth, Jackson, Gries and others who have laid the foundations of much of what is current programming would, I am sure, be the first to agree with this assertion. Methodologies, by their very nature, are always developed within a limited context and with limited experience and so we must always be wary about their mechanical application.

Much of what we have discussed in this paper is concerned with program quality. Unfortunately, quality in any creative enterprise usually has the definition of being that which is always produced by the creator. The reader must make his or her own judgements as to whether we have also fallen into this trap. Our over-riding intent has been to explore the use of tools that may be helpful, both in improving the quality of the design process and of the designed product. In attempting to develop an effective design process we have sought well-defined composition rules that can partition both the design process itself, and also the finished product. Our measures of quality in the finished product have been semantic clarity, generality, and mechanistic (as distinct from textual) simplicity. Our proposals relating to the characterization of loops have been somewhat more exploratory. However, they may provide a framework in which an intelligent and consistent discussion of program quality could take place.

References

1. Floyd, R.W., Assigning Meanings to Programs. In: Proc. Sym. in App. Math., Vol 19, Mathematical Aspects of Computer Science (J.T. Schwartz, ec.) Am. Math. Soc. 19–32

2. Hoare, C.A.R., "Proof of a Program: Find", Comm. ACM, 14, 39–45 (1971)

3. Dijkstra, E.W., "A Discipline of Programming", Prentice–Hall, Englewood Cliffs N.S. (1976)

4. Gries, D., 'The Science of Programming", Springer–Verlag, N.Y. (1981)

5. Jackson, M.A., "Principles of Program Design", Academic Press, London (1975)

6. Warnier, J.D. "Logical Construction of Programs", van Nostrand, Reinhold, N.Y. (1974)

7. Dijkstra, E.W., "The Development of Programming Methodology", in "Programming Language Systems", Eds., M.C. Newey, R.B. Stanton, G.L. Wolfendale, ANU Press, Canberra (1975)

8. Feuer, A.R., and Gehani, N.H., "A Comparison of the Programming Languages (and Pascal)", Comp. Surv. 14, 73–92 (1982)

9. Turski, W.M., "Computer Programming Methodology", Heyden, London (1978)

10. Polya, G., "Induction and Analogy in Mathematics", Princeton University Press, N.J. (1973)

11. Bergland, G.D., "A Guided Tour of Program Design Methodologies", Computer, 14, No. 10, 13–37 (1981)

12. Alagic, S., and Arbib, M.A., "The Design of Well–Structured and Correct Programs", Springer–Verlag, N.Y. (1978)

13. Dromey, R.G., "How to Solve It by Computer", Prentice–Hall, London (1982)

14. Dijkstra, E.W., " Notes on Structured Programming " in: O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, "Structured Programming", Academic Press, London (1972)

15. Knuth, D.E., "Structured Programming with goto Statements", Comp. Surv., 6, 261–301 (1974)

16. Wirth, N. "Algorithms + Data Structures = Programs", Prentice–Hall, Englewood Cliffs N.J. (1976)

17. Knuth, D.E., "The Art of Computer Programming: Vol 1/ Fundamental Algorithms", Addison-Wesley, N.Y. (1968)

18. Chomsky, N., "Syntactic Structures", Mouton & Co., The Hague (1957)

19. Chomsky, N., "Aspects of the Theory of Syntax", M.I.T. Press, Cambridge, Mass (1965)

20. Bauer, F., and J. Eickel (Eds.), "Compiler Construction" Springer-Verlag, Heidelberg, (1976)

Acknowledgements

I would like to thank C. Bailes, P. Bailes,, G. Doherty, R.F. Hille, P. Maker, M. Milway, R. McConchie, R. Nealon, I.G. Pirie, J. Reinfelds, A. Salvadori, J. Schuster, and P. Strazdins, for helpful discussions in the course of this work.

I would also especially like to thank Lynn Maxwell for her excellent work in preparing the manuscript.

DTIC

END

4-86